Table of Contents



Flask Web 开发入门

version 1.1 License CC BY-NC-ND 4.0 analytics GA

Flask 简介

Python 中有许多 Web 开发框架，比如 Django，Flask，Tornado，Bottle 和 web.py 等，其中，Django 可以说是一个全能型（all in one）的框架，自带管理后台；而 Flask 则是一个非常轻量级的框架，提供了搭建 Web 服务的必要组件，如果你不喜欢自带的组件，由于 Flask 良好的扩展性，你也可以使用其他开源的 Flask 扩展插件，甚至可以自己写一个，让喜欢折腾的开发者一展身手；Tornado 则主打异步处理，高并发，这也是它的一个显著特点。

第一次接触到 Flask 时被它的简洁感动了，几行代码就可以快速搭建出一个简单的 Web 服务，于是就义无反顾地踏上了 Flask 的学习之路，慢慢地就学习到了诸如 Jinja2 模板引擎，路由，视图，静态文件和蓝图等。Flask 非常小，源码文件包括注释在内，总共才 6000 多行，当你能熟练使用 Flask 的各个模块时，相信你也可以读懂它的所有源码。

关于本书

本书的写作开始于 2016 年 7 月，当时的初衷就是想把学的东西记录下来，但是比较分散，后来想到可以把它写成一本开源的电子书，何乐而不为？可是真正写的时候，才发现写书真的好费精力。但不管怎样，最后还是写了一些东西。9 月份发布

了第 1 版，收到不少网友的良好建议，所以又抽空进行了完善，当然，也拖了不少时间。

本书主要介绍 Flask 的基本使用，这也是我一开始在学习 Flask 过程中经常用到的。我也希望读者能通过本书快速掌握 Flask 的基本功能，快速构建出自己的 Web 服务。阅读本书可能需要读者掌握基本的 Python 语法知识，以及简单的 HTML 语法。

本书主要分为五个章节：

- 第 1 章：介绍 Flask 的安装和快速使用。
- 第 2 章：介绍 Flask 的基本使用方法，比如路由和视图，静态模板，蓝图和工厂方法等。
- 第 3 章：介绍 Flask 常用扩展插件的使用方法。
- 第 4 章：Flask 实战，介绍了如何开发一个 Web TODO 应用。
- 第 5 章：结束语，包含一些相关的参考资料以及资源推荐。

声明



更新记录

时间	说明
2016-11-14	发布版本 v1.1，增加了蓝图、工厂方法、消息闪现和 Flask 常用扩展等
2016-09-10	发布版本 v1.0，包含基本的路由和视图，模板引擎，部署等
2016-08-22	基本完成初稿

联系我

如果你对于本书有什么建议或意见，欢迎批评指正，并联系我。

- 个人主页
- GitHub
- Twitter

支持我

如果你觉得本书对你有所帮助，不妨请我喝杯咖啡，感谢支持！




支付宝
ALIPAY



基本安装

Flask 的安装很简单，可以全局安装，也可以使用虚拟环境安装。

全局安装

全局安装可以直接使用以下命令：

```
$ sudo pip install flask
```

使用 virtualenvwrapper

- 第 1 步，先安装 virtualenvwrapper，`$ [sudo] pip install virtualenvwrapper`
- 第 2 步，`$ source /usr/local/bin/virtualenvwrapper.sh`
- 第 3 步，新建一个虚拟环境，`$ mkvirtualenv env1`

此时，可以看到命令行前面会多出一个括号，这说明你已经进入名为 `env1` 的虚拟环境了，以后 `easy_install` 或者 `pip` 安装的所有模块都会安装到该虚拟环境目录里。

- 第 4 步，安装 flask

```
(env1) $ pip install flask
```

一个最简单的例子：Hello World

新建一个脚本文件，比如 `hello.py`。

```
$ cat hello.py

from flask import Flask

app = Flask(__name__)

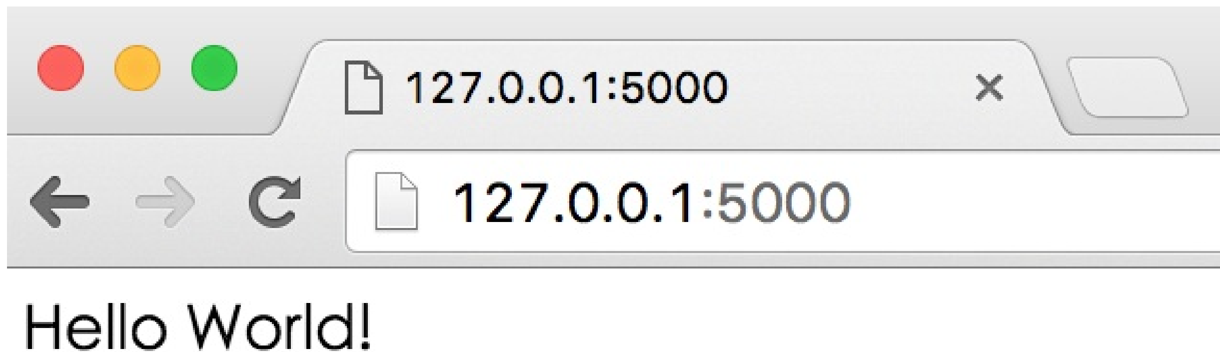
@app.route("/")
def index():
    return "Hello World!"

if __name__ == "__main__":
    app.run()
```

在终端运行：

```
$ python hello.py
* Running on http://localhost:5000/
```

在浏览器输入链接 `http://localhost:5000/` ，可以看到 Hello World!



快速入门

本章主要介绍 flask 的基础使用，主要包含以下几个方面：

- 路由和视图
- 静态文件
- Jinja2 模板引擎
- 请求、重定向及会话
- 消息闪现
- 蓝图
- 工厂方法
- REST Web 服务
- 部署

一个最简单的应用

我们在第 1 章已经看到了一个简单的 Hello World 的例子，相信你已经成功地把它跑起来了，下面我们对这个程序进行讲解。回顾一下这个程序：

```
$ cat hello.py

from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run()
```

- 先看程序的第 1 句：

```
from flask import Flask
```

该句从 flask 包导入了一个 Flask 类，这也是后面构建 Flask Web 程序的基础。

- 接着看程序的第 2 句：

```
app = Flask(__name__)
```

上面这一句通过将 `__name__` 参数传给 Flask 类的构造函数，创建了一个程序实例 `app`，也就创建了一个 Flask 集成的开发 Web 服务器。**Flask** 用 **`__name__`** 这个参数决定程序的根目录，以便程序能够找到相对于程序根目录的资源文件位置，比如静态文件等。

- 接着看程序的第 3，4，5 句：

```
@app.route("/")
def hello():
    return "Hello World!"
```

可能读者会对这三句感到很困惑：它们的作用是什么呢？我们知道，Web 浏览器把请求发送给 Web 服务器，Web 服务器再把请求发送给 Flask 程序实例，那么程序实例就需要知道对每个 URL 请求应该运行哪些代码。

上面这三句代码的意思就是：如果浏览器要访问服务器程序的根地址（"/"），那么 Flask 程序实例就会执行函数 `hello()` ，返回『Hello World!』。

比如，假设我们部署程序的服务器域名为 `www.hello.com` ，当我们在浏览器访问 http:// www.hello.com（也就是根地址）时，会触发 Flask 程序执行 `hello()` 这个函数，返回『Hello World!』，这个函数的返回值称为响应，是客户端接收到的内容。

但是，如果我们在浏览器访问 http://www.hello.com/peter 时，程序会返回 `404` 错误，因为我们的 Flask 程序并没有对这个 URL 指定处理函数，所以会返回错误代码。

- 接着看程序的最后两句：

```
if __name__ == "__main__":
    app.run()
```

上面两句的意思，当我们运行该脚本的时候（第 1 句），启动 Flask 集成的开发 Web 服务器（第 2 句）。默认情况下，改服务器会监听本地的 5000 端口，如果你想改变端口的话，可以传入 "port=端口号"，另外，如果你想支持远程，需要传入 "host=0.0.0.0"，你还可以设置调试模式，如下：

```
app.run(host='0.0.0.0', port=8234, debug=True)
```

服务器启动后，程序会进入轮询，等待并处理请求。轮询会一直运行，直到程序被终止。需要注意的是，Flask 提供的 Web 服务器不适合在生产环境中使用，后面我们会介绍生产环境中的 Web 服务器。

OK，到此为止，我们基本明白一个简单的 Flask 程序是怎么运作的了，后面我们就一起慢慢揭开 Flask 的神秘面纱吧~~

路由和视图

我们在前面的一小节介绍了一个简单的 Flask 程序是怎么运行的。其中，有三行代码，我们并没有深入讲解。在这里，我们就对它们进行深入解析。回顾这三行代码：

```
@app.route("/")
def hello():
    return "Hello World!"
```

这三行代码的意思就是：如果浏览器要访问服务器程序的根地址（"/"），那么 Flask 程序实例就会执行函数 `hello()` ，返回『Hello World!』。

也就是说，上面三行代码定义了一个 **URL** 到 **Python** 函数的映射关系，我们将处理这种映射关系的程序称为『路由』，而 `hello()` 就是视图函数。

动态路由

假设服务器域名为 `https://hello.com`，我们来看下面一个路由：

```
@app.route("/ethan")
def hello():
    return '<h1>Hello, ethan!</h1>'
```

再来看一个路由：

```
@app.route("/peter")
def hello():
    return '<h1>Hello, peter!</h1>'
```

可以看到，上面两个路由的功能是当用户访问 `https://hello.com/<user_name>` 时，网页显示对该用户的问候。按上面的写法，如果对每个用户都需要写一个路由，那么 100 个用户岂不是要写 100 个路由！这当然是不能忍受的，实际上一个路由就够了！且看下面：

```
@app.route("/<user_name>")
def hello(user_name):
    return '<h1>Hello, %s!</h1>' % user_name
```

现在，任何类似 `https://hello.com/<user_name>` 的 URL 都会映射到这个路由上，比如 `https://hello.com/ethan-funny`，`https://hello.com/torvalds`，访问这些 URL 都会执行上面的路由程序。

也就是说，Flask 支持这种动态形式的路由，路由中的动态部分默认是字符串，像上面这种情况。当然，除了字符串，Flask 也支持在路由中使用 int、float，比如路由 `/articles/<int:id>` 只会匹配动态片段 id 为整数的 URL，例如匹配 https://hello.com/articles/100，https://hello.com/articles/101，但不匹配 https://hello.com/articles/the-first-article 这种 URL。

静态文件

静态文件，顾名思义，就是那些不会被改变的文件，比如图片，CSS 文件和 JavaScript 源码文件。默认情况下，Flask 在程序根目录中名为 static 的子目录中寻找静态文件。因此，我们一般在应用的包中创建一个叫 static 的文件夹，并在里面放置我们的静态文件。比如，我们可以按下面的结构组织我们的 app：

```
app/
    __init__.py
    static/
        css/
            style.css
            home.css
            admin.css
        js/
            home.js
            admin.js
        img/
            favicon.co
            logo.svg
    templates/
        index.html
        home.html
        admin.html
    views/
    models/
run.py
```

但是，我们有时还会应用到第三方库，比如 jQuery, Bootstrap 等，这时我们为了不跟自己的 Javascript 和 CSS 文件混起来，我们可以将这些第三方库放到 lib 文件夹或者 vendor 文件夹，比如下面这种：

```
static/
    css/
        lib/
            bootstrap.css
        style.css
        home.css
        admin.css
    js/
        lib/
            jquery.js
            chart.js
        home.js
        admin.js
    img/
        logo.svg
        favicon.ico
```

提供一个 favicon 图标

favicon 是 favorites icon 的缩写，也被称为 website icon（网页图标）、page icon（页面图标）等。通常而言，定义一个 favicon 的方法是将一个名为『favicon.ico』的文件置于 Web 服务器的根目录下。但是，正如我们在上面指出，我们一般将图片等静态资源放在一个单独的 static 文件夹中。为了解决这种不一致，我们可以在站点模板的 部分添加两个 link 组件，比如我们可以在 template/base.html 中定义 favicon 图标：

```
{% block head %}
{{ super() }}
<link rel="shortcut icon" href="{{ url_for('static', filename =
'favicon.ico') }}" type="image/x-icon">
<link rel="icon" href="{{ url_for('static', filename = 'favicon.
ico') }}" type="image/x-icon">
{% endblock %}
```

在上面的代码中，我们使用了 `super()` 来保留基模板中定义的块的原始内容，并添加了两个 link 组件声明图标位置，这两个 link 组件声明会插入到 head 块的末尾。

使用 Jinja2 模板引擎

什么是模板引擎

在 Web 开发中，我们经常会使用到模板引擎。简单点来说，我们可以把模板看成是一个含有某些变量的字符串，它们的具体值需要在动态运行时（请求的上下文）才能知道。比如，有下面一个模板：

```
<h1>Hello, {{ name }}!</h1>
```

其中，name 是一个变量名，我们用 `{{ }}` 包裹它表示它是一个变量。我们给 name 传不同的值，模板会返回不同的字符串。像这样，使用真实的值替换变量，再返回最终得到的响应字符串，这一过程称为渲染。模板引擎就是渲染模板的程序。

Flask 默认使用 Jinja2 模板引擎。

为什么要使用模板引擎

先来看一个简单的程序。

```
$ cat hello.py

from flask import Flask

app = Flask(__name__)

@app.route("/<name>")
def hello(name):
    if name == 'ethan':
        return "<h1>Hello, world!</h1> <h2>Hello, %s!</h2>" % na
me
    else:
        return "<h1>Hello, world!</h1> <h2>Hello, world!</h2>"

if __name__ == "__main__":
    app.run()
```

在终端运行上面的代码 `python hello.py` ，终端输出：

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

我们在浏览器地址栏输入 `http://127.0.0.1:5000/ethan` ，显示如下：



符合预期，没什么问题。但是，我们看到，上面的视图函数 `hello()` 夹杂了一些 `HTML` 代码，如果 HTML 代码多了，会使得我们的程序变得难以理解和维护。我们可以看到，其实视图函数主要有两部分逻辑：业务逻辑和表现逻辑。像上下文判断，数据库查询等后台处理都可以算是业务逻辑，而返回给前端的响应内容则算是表现逻辑，它们是需要在前端展现的。

在上面的代码中，我们将业务逻辑和表现逻辑混杂在一起了，代码很不优雅，而且当代码变多了后，程序将会变得难以理解和维护。因此，良好的做法应该是将业务逻辑和表现逻辑分开，模板引擎正好可以满足这种需求。

Jinja 模板引擎入门

我们将上面的例子用 Jinja 模板进行改写。默认情况下，Flask 在程序文件夹中的 templates 子文件夹中寻找模板。改写后的文件结构如下：

```
.
├── hello.py
└── templates
    └── index.html
```

`hello.py` 文件内容如下：

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/<name>')
def hello(name):
    if name == 'ethan':
        return render_template('index.html', name=name)
    else:
        return render_template('index.html', name='world')

if __name__ == "__main__":
    app.run()
```

`index.html` 文件内容如下：

```
<h1>Hello, world!</h1> <h2>Hello, {{ name }}!</h2>
```

在 `hello.py` 中，我们使用了Flask 提供的 render_template 函数，该函数把 Jinja2 模板引擎集成到了程序中。render_template 函数的第一个参数是模板的文件名。随后的参数都是键值对，表示模板中变量对应的真实值。

变量

Jinja 模板使用 `{{ 变量名 }}` 表示一个变量，比如上面的 `{{ name }}`，它告诉模板引擎这个位置的值从渲染模板时使用的数据中获取。

在 Jinja 中，还能使用列表，字典和对象等复杂的类型，比如：

```
<p> Hello, {{ mydict['key'] }}. Hello, {{ mylist[0] }}. </p>
```

控制结构

Jinja 提供了多种控制结构，来改变模板的渲染流程，比如常见的判断结构，循环结构。示例如下：

```
{% if user == 'ethan' or user =='peter' %}
    <p> Hello, {{ user }} </p>
{% else %}
    <p> Hello, world! </p>
{% endif %}

<ul>
    {% for user in user_list %}
        <li> {{ user }} </li>
    {% endfor %}
</ul>
```

宏

当有一段代码我们经常要用到的时候，我们往往会写一个函数，在 Jinja 中，我们可以使用宏来实现。例如:

```
{% macro render_user(user) %}
    {% if user == 'ethan' %}
        <p> Hello, {{ user }} </p>
    {% endif %}
{% endmacro %}
```

为了重复使用宏，我们将其保存在单独的文件中，比如 'macros.html'，然后在需要使用的模板中导入：

```
{% import 'macros.html' as macros %}

{{ macros.render_user(user) }}
```

模板继承

另一种重复使用代码的强大方式是模板继承，就像类继承需要有一个基类一样，我们需要一个基模板。比如，我们可以创建一个名为 base.html 的基模板：

```
<html>
<head>
    {% block head %}
    <title>{% block title %}{% endblock %} - My Application</tit
le>
    {% endblock %}
</head>

<body>
    {% block body %}
    {% endblock %}
</body>
</html>
```

我们可以看到上面的基模板含有三个 `block` 块：head、title 和 body。下面，我们通过这个基模板来派生新的模板：

```
{% extends "base.html" %}
{% block title %}Index{% endblock %}
{% block head %}
    {{ super() }}
{% endblock %}
{% block body %}
<h1>Hello, World!</h1>
{% endblock %}
```

注意到上面第一行代码使用了 `extends` 命令，表明该模板继承自 base.html。接着，我们重新定义了 title、head 和 body。另外，我们还使用了 super() 获取基模板原来的内容。

更多关于 Jinja 模板引擎的使用可以参考 Jinja2 2.7 documentation。

请求、重定向及会话

Web 开发中经常需要处理 HTTP 请求、重定向和会话等诸多事务，相应地，Flask 也内建了一些常见的对象如 request, session, redirect 等对它们进行处理。

请求对象 request

HTTP 请求方法有 GET、POST、PUT 等，request 对象也相应地提供了支持。举个例子，假设现在我们开发一个功能：用户注册。如果 HTTP 请求方法是 POST，我们就注册该用户，如果是 GET 请求，我们就显示注册的字样。代码示例如下（注意，下面代码并不能直接运行，文末提供了完整的代码）：

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/register', methods=['POST', 'GET']):
def register():
    if request.method == 'GET':
        return 'please register!'
    elif request.method == 'POST':
        user = request.form['user']
        return 'hello', user
```

重定向对象 redirect

当用户访问某些网页时，如果他还没登录，我们往往会把网页重定向到登录页面，Flask 提供了 redirect 对象对其进行处理，我们对上面的代码做一点简单的改造，如果用户注册了，我们将网页重定向到首页。代码示例如下：

```
from flask import Flask, request, redirect

app = Flask(__name__)

@app.route('/home', methods=['GET']):
def index():
    return 'hello world!'

@app.route('/register', methods=['POST', 'GET']):
def register():
    if request.method == 'GET':
        return 'please register!'
    elif request.method == 'POST':
        user = request.form['user']
        return redirect('/home')
```

会话对象 session

程序可以把数据存储在用户会话中，用户会话是一种私有存储，默认情况下，它会保存在客户端 cookie 中。Flask 提供了 session 对象来操作用户会话，下面看一个示例：

```
from flask import Flask, request, session, redirect, url_for, re
nder_template

app = Flask(__name__)

@app.route('/home', methods=['GET'])
def index():
    return 'hello world!'

@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        user_name = request.form['user']
        session['user'] = user_name
        return 'hello, ' + session['user']
    elif request.method == 'GET':
        if 'user' in session:
            return redirect(url_for('index'))
        else:
            return render_template('login.html')

app.secret_key = '123456'

if __name__ == '__main__':
    app.run(host='127.0.0.1', port=5632, debug=True)
```

操作 `session` 就像操作 python 中的字典一样，我们可以使用 `session['user']` 获取值，也可以使用 `session.get('user')` 获取值。注意到，我们使用了 `url_for` 生成 URL，比如 `/home` 写成了 `url_for('index')` 。`url_for()` 函数的第一个且唯一必须指定的参数是端点名，即路由的内部名字。默认情况下，路由的端点是相应视图函数的名字，因此 `/home` 应该写成 `url_for('index')` 。还有一点，使用 `session` 时要设置一个密钥 `app.secret_key` 。

附录

本节完整的代码如下：

```
$ tree .
```

```
.
├── flask-session.py
└── templates
    ├── layout.html
    └── login.html

$ cat flask-session.py
from flask import Flask, request, session, redirect, url_for, re
nder_template

app = Flask(__name__)

@app.route('/')
def head():
    return redirect(url_for('register'))

@app.route('/home', methods=['GET'])
def index():
    return 'hello world!'

@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        user_name = request.form['user']
        session['user'] = user_name
        return 'hello, ' + session['user']
    elif request.method == 'GET':
        if 'user' in session:
            return redirect(url_for('index'))
        else:
            return render_template('login.html')

app.secret_key = '123456'

if __name__ == '__main__':
    app.run(host='127.0.0.1', port=5632, debug=True)

$ cat layout.html
<!doctype html>
<title>Hello Sample</title>
<link rel="stylesheet" type="text/css" href="{{ url_for('static'
, filename='style.css') }}">
```

```
<div class="page">
    {% block body %}
    {% endblock %}
</div>

$ cat login.html
{% extends "layout.html" %}
{% block body %}
<form name="register" action="{{ url_for('register') }}" method=
"post">
    Hello {{ title }}, please login by:
    <input type="text" name="user" />
</form>
{% endblock %}
```

蓝图

在前面，我们都是把代码写在单一的文件里面，虽然看起来很方便，但也只是供学习的时候用用而已，真正在一个实际项目中，是不应该这样做的，为什么呢？

我们还是从 `hello world` 开始讲起，新建一个脚本文件，比如 `hello.py`。

```
$ cat hello.py

# -*- coding: utf-8 -*-

from flask import Flask

app = Flask(__name__)

@app.route("/")
def index():
    return "Hello World!"

if __name__ == "__main__":
    app.run(host='127.0.0.1', port=5200, debug=True)
```

运行该脚本，在浏览器输入链接 `http://localhost:5200/`，可以看到 `Hello World!` 的字样。

OK，现在我们需要添加一个『读书』频道，允许查看并添加书籍。这不很简单吗，将上面的代码修改一下，如下：

```
$ cat hello.py

# -*- coding: utf-8 -*-

from flask import Flask, request, redirect, url_for

app = Flask(__name__)
app.secret_key = 'some secret key'

books = ['the first book', 'the second book', 'the third book']
```

```
@app.route("/")
def index():
    render_string = '<ul>'

    for book in books:
        render_string += '<li>' + book + '</li>'

    render_string += '</ul>'

    return render_string

@app.route("/book", methods=['POST', 'GET'])
def book():
    _form = request.form

    if request.method == 'POST':
        title = _form["title"]
        books.append(title)
        return redirect(url_for('index'))

    return '''
        <form name="book" action="/book" method="post">
            <input id="title" name="title" type="text" placehold
er="add book">
            <button type="submit">Submit</button>
        </form>
        '''

if __name__ == "__main__":
    app.run(host='127.0.0.1', port=5200, debug=True)
```

OK，上面的脚本实现了最基本的查看并添加书籍的功能。

接着，我们还想添加『电影』、『音乐』、『美食』和『旅游』等频道，我们是继续往这个文件添加功能吗？当然不是，是时候把这些功能进行拆分了，将我们的应用模块化，把一个大应用分解成若干个小应用。你可能会想：把这些功能分别写到多个文件，比如 `movie.py` ，`music.py` 等等，可是这样的话，我们的应用是跑不起来的。为什么呢？因为 `app` 对象只有一个。

那怎么办呢？

事实上，Flask 提供了 Blueprint (蓝图) 的功能，让我们可以实现模块化的应用。使用它主要有以下好处：

- 将一个复杂的大型应用分解成若干蓝图的集合，也就是若干个子应用或者说模块，每个蓝图都包含了可以作为独立模块的视图、模板和静态文件等；
- 制作通用的组件，使开发者更易复用组件；

不过目前 Flask 蓝图的注册是静态的，不支持可插拔。

现在，让我们将上面的代码进行改写，加入蓝图的功能。

首先，我们对程序的结构做一些调整，如下：

```
├── app.py                  -- 启动程序
├── book                    -- book 模块
│   ├── __init__.py
│   └── book.py
├── movie                   -- movie 模块
│   ├── __init__.py
│   └── movie.py
└── templates               -- 模板
    ├── 404.html
    ├── book.html
    ├── layout.html
    └── movie.html
```

其中，movie 模块跟 book 模块是独立的，它们有各自的业务逻辑，互不影响。我们这里只分析 book 模块，book.py 的代码如下：

```
# -*- coding: utf-8 -*-

from flask import Blueprint, url_for, render_template, request,
flash, redirect

# 创建一个蓝图对象
book_bp = Blueprint(
    'book',
    __name__,
    template_folder='../templates',
)
```

```
books = ['The Name of the Rose', 'The Historian', 'Rebecca']

@book_bp.route('/', methods=['GET'])
def index():
    return '<h1>Hello World!</h1>'

@book_bp.route('/book', methods=['GET', 'POST'])
def handle_book():
    _form = request.form

    if request.method == 'POST':
        title = _form["title"]
        books.append(title)
        flash("add book successfully!")
        return redirect(url_for('book.handle_book'))

    return render_template(
        'book.html',
        books=books
   )

@book_bp.route('/book/<name>')
def get_book_info(name):
    book = [name]
    if name not in books:
        book = []

    return render_template(
        'book.html',
        books=book
    )
```

注意到，我们使用了下面的代码创建一个蓝图对象：

```
book_bp = Blueprint('book', __name__, template_folder='../templa
tes')
```

Blueprint 要求至少传入两个参数，第一个参数是蓝图的名称，第二个参数是蓝图所在的包或模块，其他参数是可选的，比如 `template_folder` ， `url_prefix` 和 `static_folder` 等。

在蓝图中使用路由是这样的：

```
# book_bp 就是我们创建的蓝图对象
@book_bp.route('/book/<name>')
```

创建好了蓝图之后，我们还需要注册它，否则不能使用。我们在 `app.py` 中注册：

```
# -*- coding: utf-8 -*-

from flask import Flask, render_template
from book import book_bp
from movie import movie_bp

app = Flask(__name__)
app.secret_key = 'The quick brown fox jumps over the lazy dog'

# 注册蓝图
app.register_blueprint(book_bp)
app.register_blueprint(movie_bp)

@app.errorhandler(404)
def page_not_found(error):
    return render_template('404.html'), 404


if __name__ == '__main__':
    app.run(host='127.0.0.1', port=5200, debug=True)
```

本文完整的代码在这里。

在命令行中输入 `$ python app.py` ，我们就可以访问该应用了，如下：



Hello World!

127.0.0.1:5200/book

- The Name of the Rose
- The Historian
- Rebecca

add book Submit

总结

- 使用蓝图对我们的应用进行模块化
- 通过添加蓝图扩展我们的应用
- 蓝图定义完之后，还需要在应用中注册
- 可以给蓝图中的所有路由定义一个 URL 前缀 (url_prefix)

更多阅读

- 用蓝图实现模块化的应用
- 蓝图 | Flask之旅
- Flask进阶系列(六)–蓝图 – 思诚之道
- 如何理解 Flask 中的蓝图？ - 知乎专栏

消息闪现

Flask 提供了消息闪现的功能，以使我们的应用可以向用户反馈信息。比如，当用户登录失败了，我们会提醒用户名错误或者密码错误。

在 Flask 中使用消息闪现很简单。下面我们以上一篇的例子进行说明。完整的代码在这里。

首先，让我们看一下 `book.py` 的代码：

```
# -*- coding: utf-8 -*-

from flask import Blueprint, url_for, render_template, request,
flash, redirect

book_bp = Blueprint(
    'book',
    __name__,
    template_folder='../templates',
)

books = ['The Name of the Rose', 'The Historian', 'Rebecca']


@book_bp.route('/', methods=['GET'])
def index():
    return '<h1>Hello World!</h1>'


@book_bp.route('/book', methods=['GET', 'POST'])
def handle_book():
    _form = request.form

    if request.method == 'POST':
        title = _form["title"]
        books.append(title)
        flash("add book successfully!")   # 使用 flash 反馈消息
        return redirect(url_for('book.handle_book'))

    return render_template(
        'book.html',
        books=books
    )
```

注意到，用户添加完书籍的时候，我们向用户反馈『添加书籍成功』的消息：

```
flash("add book successfully!")
```

但是，还有一个步骤要做，才能让用户接收到这个消息，那就是在模板中获取这个消息，通过在 `layout.html` 中使用 `get_flashed_messages()` 获取消息， `layout.html` 代码如下：

```
<!doctype html>
<title>Hello Sample</title>
<link rel="stylesheet" type="text/css" href="{{ url_for('static'
, filename='style.css') }}">

<div class="page">
    {% block body %} {% endblock %}
</div>

{% for message in get_flashed_messages() %}
    {{ message }}
{% endfor %}
```

分类闪现消息

我们还可以对闪现消息进行分类，比如有些消息是正常的通知消息，而有些消息是出错消息。同样，我们需要两个步骤：

- 使用 flash() 函数的第二个参数，不使用的话，默认是 'message'

```
flash('add book fail', 'error')
```

- 接着，在模板中调用 `get_flashed_messages()` 函数来返回这个分类，类似下面：

```
{% with messages = get_flashed_messages(with_categories=true) %}
  {% for category, message in messages %}
    {{category}}: {{ message }}
  {% endfor %}
{% endwith %}
```

过滤闪现消息

过滤闪现消息在模板中可以这样用：

```
{% with error_messages = get_flashed_messages(category_filter=["
error"]) %}
  {% for error in error_messages %}
    {{ error }}
  {% endfor %}
{% endwith %}
```

更多阅读

- 消息闪现 — Flask 0.10.1 文档
- 浅入浅出Flask框架：flashing system | 樂天笔记
- Flask入门系列(五)–错误处理及消息闪现 – 思诚之道

工厂方法

在前面，我们都是直接通过 `app=Flask(__name__)` 来创建一个 `app` 实例的。这样做没什么问题，但如果我们想为每个实例分配不同的配置，比如有测试环境的配置，开发环境的配置和生产环境的配置等，这时就比较麻烦了。

有什么办法呢？

其实我们可以通过调用一个函数来返回一个应用实例，比如下面的方法：

```
def create_app(config_filename):
    app = Flask(__name__)
    app.config.from_pyfile(config_filename)

    from yourapplication.views.admin import admin
    from yourapplication.views.user import user
    app.register_blueprint(admin)
    app.register_blueprint(user)

    return app
```

上面的 `create_app` 函数就是一个 `工厂方法` ，我们将创建应用程序实例的工作交给了它来完成，我们以后就可以通过传入不同的配置名，以此批量生产 `app` 。

说到这，你也应该明白 `工厂方法` 的优势所在了：

- 将创建应用实例的过程交给工厂函数，通过传入不同的配置，我们可以创建不同环境下的应用
- 在做测试时，为每个实例分配分配不同的配置，从而测试每一种不同的情况

现在，我们对上一篇的例子进行重构，引入工厂函数。

核心代码在下面，完成代码请参考这里。

现在， `app.py` 代码如下：

```
# -*- coding: utf-8 -*-

from flask import Flask, render_template
from configs import config
from book import book_bp
from movie import movie_bp

def create_app(config_name):
    app = Flask(__name__)

    # basic config
    app.config.from_object(config[config_name])

    # blueprints
    app.register_blueprint(book_bp)
    app.register_blueprint(movie_bp)

    # error handler
    handle_errors(app)

    return app

def handle_errors(app):
    @app.errorhandler(404)
    def page_not_found(error):
        return render_template('404.html'), 404
```

我们还需要一个启动程序，比如 `run.py` ，如下：

```
# -*- coding: utf-8 -*-

from app import create_app

if __name__ == '__main__':
    app = create_app('default')
    app.run(host='127.0.0.1', port=5200, debug=True)
```

打开终端，运行下面命令，就可以在浏览器访问我们的应用了。

```
$ python run.py
```

更多阅读

- 应用程序的工厂函数 — Flask 0.10.1 文档
- Flask进阶系列(七)–应用最佳实践 – 思诚之道

REST Web 服务

什么是 REST

REST 全称是 Representational State Transfer，翻译成中文是『表现层状态转移』，估计读者看到这个词也是云里雾里的，我当初也是！这里，我们先不纠结这个词到底是什么意思。事实上，REST 是一种 Web 架构风格，它有六条准则，满足下面六条准则的 Web 架构可以说是 Restuful 的。

1. 客户端-服务器（Client-Server）

   服务器和客户端之间有明确的界限。一方面，服务器端不再关注用户界面和用户状态。另一方面，客户端不再关注数据的存储问题。这样，服务器端跟客户端可以独立开发，只要它们共同遵守约定。

2. 无状态（Stateless）

   来自客户端的每个请求必须包含服务器所需要的所有信息，也就是说，服务器端不存储来自客户端的某个请求的信息，这些信息应由客户端负责维护。

3. 可缓存（Cachable）

   服务器的返回内容可以在通信链的某处被缓存，以减少交互次数，提高网络效率。

4. 分层系统（Layered System）

   允许在服务器和客户端之间通过引入中间层（比如代理，网关等）代替服务器对客户端的请求进行回应，而且这些对客户端来说不需要特别支持。

5. 统一接口（Uniform Interface）

   客户端和服务器之间通过统一的接口（比如 GET, POST, PUT, DELETE 等）相互通信。

6. 支持按需代码（Code-On-Demand，可选）

   服务器可以提供一些代码（比如 Javascript）并在客户端中执行，以扩展客户端的某些功能。

使用 Flask 提供 REST Web 服务

REST Web 服务的核心概念是资源（resources）。资源被 URI（Uniform Resource Identifier, 统一资源标识符）定位，客户端使用 HTTP 协议操作这些资源，我们用一句不是很全面的话来概括就是：URI 定位资源，用 HTTP 动词（GET, POST, PUT, DELETE 等）描述操作。下面列出了 REST 架构 API 中常用的请求方法及其含义：

HTTP Method	Action	Example
GET	从某种资源获取信息	http://example.com/api/articles (获取所有文章)
GET	从某个资源获取信息	http://example.com/api/articles/1 (获取某篇文章)
POST	创建新资源	http://example.com/api/articles (创建新文章)
PUT	更新资源	http://example.com/api/articles/1 (更新文章)
DELETE	删除资源	http://example.com/api/articels/1 (删除文章)

设计一个简单的 Web Service

现在假设我们要为一个 blog 应用设计一个 Web Service。

首先，我们先明确访问该 Service 的根地址是什么。这里，我们可以这样定义：

```
http://[hostname]/blog/api/
```

然后，我们明确有哪些资源是要公开的。可以知道，我们这个 blog 应用的资源就是 articles。

下一步，我们要明确怎么去操作这些资源，如下所示：

HTTP Method	URI	Action
GET	http://[hostname]/blog/api/articles	获取所有文章列表
GET	http://[hostname]/blog/api/articles/[article_id]	获取某篇文章内容
POST	http://[hostname]/blog/api/articles	创建一篇新的文章
PUT	http://[hostname]/blog/api/articles/[article_id]	更新某篇文章
DELETE	http://[hostname]/blog/api/articles/[article_id]	删除某篇文章

为了简便，我们定义一篇 article 的属性如下：

- id：文章的 id，Numeric 类型
- title: 文章的标题，String 类型
- content: 文章的内容，TEXT 类型

至此，我们基本完成了这个 Web Service 的设计，下面我们就来实现它。

使用 Flask 提供 RESTful api

在实现这个 Web 服务之前，我们还有一个问题没有考虑到：我们应该怎么存储我们的数据。毫无疑问，我们应该使用数据库，比如 MySql、MongoDB 等。但是，数据库的存储不是我们这里要讨论的重点，所以我们采用一种偷懒的做法：使用一个内存中的数据结构来代替数据库。

GET 方法

下面我们使用 GET 方法获取资源。

```
# -*- coding: utf-8 -*-

from flask import Flask, jsonify, abort, make_response

app = Flask(__name__)

articles = [
    {
        'id': 1,
        'title': 'the way to python',
```

```
            'content': 'tuple, list, dict'
        },
        {
            'id': 2,
            'title': 'the way to REST',
            'content': 'GET, POST, PUT'
        }
]

@app.route('/blog/api/articles', methods=['GET'])
def get_articles():
    """ 获取所有文章列表 """
    return jsonify({'articles': articles})

@app.route('/blog/api/articles/<int:article_id>', methods=['GET'
])
def get_article(article_id):
    """ 获取某篇文章 """
    article = filter(lambda a: a['id'] == article_id, articles)
    if len(article) == 0:
        abort(404)

    return jsonify({'article': article[0]})

@app.errorhandler(404)
def not_found(error):
    return make_response(jsonify({'error': 'Not found'}), 404)


if __name__ == '__main__':
    app.run(host='127.0.0.1', port=5632, debug=True)
```

将上面的代码保存为文件 `app.py`，通过 `python app.py` 启动这个 Web Service。

接下来，我们进行测试。这里，我们采用命令行语句 curl 进行测试。

开启终端，敲入如下命令进行测试：

```
$ curl -i http://localhost:5632/blog/api/articles
HTTP/1.0 200 OK
Content-Type: application/json
```

```
Content-Length: 224
Server: Werkzeug/0.11.4 Python/2.7.11
Date: Tue, 16 Aug 2016 15:21:45 GMT

{
  "articles": [
    {
      "content": "tuple, list, dict",
      "id": 1,
      "title": "the way to python"
    },
    {
      "content": "GET, POST, PUT",
      "id": 2,
      "title": "the way to REST"
    }
  ]
}


$ curl -i http://localhost:5632/blog/api/articles/2
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 101
Server: Werkzeug/0.11.4 Python/2.7.11
Date: Wed, 17 Aug 2016 02:37:48 GMT

{
  "article": {
    "content": "GET, POST, PUT",
    "id": 2,
    "title": "the way to REST"
  }
}


$ curl -i http://localhost:5632/blog/api/articles/3
HTTP/1.0 404 NOT FOUND
Content-Type: application/json
Content-Length: 26
Server: Werkzeug/0.11.4 Python/2.7.11
Date: Wed, 17 Aug 2016 02:32:10 GMT
```

```
{
  "error": "Not found"
}
```

上面，我们分别测试了『获取所有文章列表』、『获取某篇文章』和『获取不存在的文章』这三个功能，结果也正是我们所预料的。

POST 方法

下面我们使用 POST 方法创建一个新的资源。在上面的代码中添加以下代码：

```
from flask import request

@app.route('/blog/api/articles', methods=['POST'])
def create_article():
    if not request.json or not 'title' in request.json:
        abort(400)
    article = {
        'id': articles[-1]['id'] + 1,
        'title': request.json['title'],
        'content': request.json.get('content', '')
    }
    articles.append(article)
    return jsonify({'article': article}), 201
```

测试如下：

```
$ curl -i -H "Content-Type: application/json" -X POST -d '{"titl
e":"the way to java"}' http://localhost:5632/blog/api/articles
HTTP/1.0 201 CREATED
Content-Type: application/json
Content-Length: 87
Server: Werkzeug/0.11.4 Python/2.7.11
Date: Wed, 17 Aug 2016 03:07:14 GMT

{
  "article": {
    "content": "",
    "id": 3,
    "title": "the way to java"
  }
}
```

可以看到，创建一篇新的文章也是很简单的。request.json 保存了请求中的 JSON 格式的数据。如果请求中没有数据，或者数据中没有 title 的内容，我们将会返回一个 "Bad Request" 的 400 错误。如果数据合法（必须要有 title 的字段），我们就会创建一篇新的文章。

PUT 方法

下面我们使用 PUT 方法更新文章，继续添加代码：

```
@app.route('/blog/api/articles/<int:article_id>', methods=['PUT'
])
def update_article(article_id):
    article = filter(lambda a: a['id'] == article_id, articles)
    if len(article) == 0:
        abort(404)
    if not request.json:
        abort(400)

    article[0]['title'] = request.json.get('title', article[0]['
title'])
    article[0]['content'] = request.json.get('content', article[0
]['content'])

    return jsonify({'article': article[0]})
```

测试如下：

```
$ curl -i -H "Content-Type: application/json" -X PUT -d '{"conte
nt": "hello, rest"}' http://localhost:5632/blog/api/articles/2
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 98
Server: Werkzeug/0.11.4 Python/2.7.11
Date: Wed, 17 Aug 2016 03:44:09 GMT

{
  "article": {
    "content": "hello, rest",
    "id": 2,
    "title": "the way to REST"
  }
}
```

可以看到，更新文章也是很简单的，上面我们更新了第 2 篇文章的内容。

DELETE 方法

下面我们使用 DELETE 方法删除文章，继续添加代码：

```
@app.route('/blog/api/articles/<int:article_id>', methods=['DELE
TE'])
def delete_article(article_id):
    article = filter(lambda t: t['id'] == article_id, articles)
    if len(article) == 0:
        abort(404)
    articles.remove(article[0])
    return jsonify({'result': True})
```

测试如下：

```
$ curl -i -H "Content-Type: application/json" -X DELETE http://l
ocalhost:5632/blog/api/articles/2
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 20
Server: Werkzeug/0.11.4 Python/2.7.11
Date: Wed, 17 Aug 2016 03:46:04 GMT

{
  "result": true
}

$ curl -i http://localhost:5632/blog/api/articles
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 125
Server: Werkzeug/0.11.4 Python/2.7.11
Date: Wed, 17 Aug 2016 03:46:09 GMT

{
  "articles": [
    {
      "content": "tuple, list, dict",
      "id": 1,
      "title": "the way to python"
    }
  ]
}
```

附录

常见 HTTP 状态码

HTTP 状态码主要有以下几类：

- 1xx —— 元数据
- 2xx —— 正确的响应
- 3xx —— 重定向
- 4xx —— 客户端错误
- 5xx —— 服务端错误

常见的 HTTP 状态码可见以下表格：

代码	说明
100	Continue。客户端应当继续发送请求。
200	OK。请求已成功，请求所希望的响应头或数据体将随此响应返回。
201	Created。请求成功，并且服务器创建了新的资源。
301	Moved Permanently。请求的网页已永久移动到新位置。 服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动将请求者转到新位置。
302	Found。服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。
400	Bad Request。服务器不理解请求的语法。
401	Unauthorized。请求要求身份验证。 对于需要登录的网页，服务器可能返回此响应。
403	Forbidden。服务器拒绝请求。
404	Not Found。服务器找不到请求的网页。
500	Internal Server Error。服务器遇到错误，无法完成请求。

curl 命令参考

选项	作用
-X	指定 HTTP 请求方法，如 POST，GET, PUT
-H	指定请求头，例如 Content-type:application/json
-d	指定请求数据
--data-binary	指定发送的文件
-i	显示响应头部信息
-u	指定认证用户名与密码
-v	输出请求头部信息

完整代码

本文的完整代码可在 Gist 查看。

参考链接

- 理解本真的REST架构风格
- 基于 Flask 实现 RESTful API | Ross's Page
- (译)使用Flask实现RESTful API - nummy的专栏 - SegmentFault
- 怎样用通俗的语言解释什么叫 REST，以及什么是 RESTful？ - 知乎
- What Does RESTful Really Mean? - DZone Integration
- HTTP状态码 - 维基百科，自由的百科全书
- 理解RESTful架构 - 阮一峰的网络日志

部署

我们这里以项目 flask-todo-app 为例，介绍如何将其部署到生产环境，主要有以下几个步骤：

- 创建项目的运行环境
- 使用 Gunicorn 启动 flask 程序
- 使用 supervisor 管理服务器进程
- 使用 Nginx 做反向代理

创建项目的运行环境

- 创建 Python 虚拟环境，以便隔离不同的项目
- 安装项目依赖包

```
$ pip install virtualenvwrapper
$ source /usr/local/bin/virtualenvwrapper.sh
$ mkvirtualenv flask-todo-env   # 创建完后，会自动进入到该虚拟环境，以后可以使用 workon 命令
$
(flask-todo-env)$ git clone https://github.com/ethan-funny/flask-todo-app
(flask-todo-env)$ cd flask-todo-app
(flask-todo-env)$ pip install -r requirements.txt
```

使用 Gunicorn 启动 flask 程序

我们在本地调试的时候经常使用命令 `python manage.py runserver` 或者 `python app.py` 等启动 Flask 自带的服务器，但是，Flask 自带的服务器性能无法满足生产环境的要求，因此这里我们采用 Gunicorn 做 wsgi (Web Server Gateway Interface，Web 服务器网关接口) 容器，假设我们以 root 用户身份进行部署：

```
(flask-todo-env)$ pip install gunicorn
(flask-todo-env)$ /home/root/.virtualenvs/flask-todo-env/bin/gunicorn -w 4 -b 127.0.0.1:7345 application.app:create_app()
```

上面的命令中，-w 参数指定了 worker 的数量，-b 参数绑定了地址（包含访问端口）。

需要注意的是，由于我们这里将 Gunicorn 绑定在本机 127.0.0.1，因此它仅仅监听来自服务器自身的连接，也就是我们无法从外网访问该服务。在这种情况下，我们通常使用一个反向代理来作为外网和 Gunicorn 服务器的中介，而这也是推荐的做法，接下来也会介绍如何使用 nginx 做反向代理。不过，有时为了调试方便，我们可能需要从外网发送请求给 Gunicorn，这时我们可以让 Gunicorn 绑定 0.0.0.0，这样它就会监听来自外网的所有请求。

使用 supervisor 管理服务器进程

在上面，我们手动使用命令启动了 flask 程序，当程序挂掉的时候，我们又要再启动一次。另外，当我们想关闭程序的时候，我们需要找到 pid 进程号并 kill 掉。这里，我们采用一种更好的方式来管理服务器进程，我们将 supervisor 安装全局环境下，而不是在当前的虚拟环境：

```
$ pip install supervisor
$ echo_supervisord_conf > supervisor.conf   # 生成 supervisor 默认配置文件
$ vi supervisor.conf    # 修改 supervisor 配置文件，添加 gunicorn 进程管理
```

在 `supervisor.conf` 添加以下内容：

```
[program:flask-todo-env]
directory=/home/root/flask-todo-app
command=/home/root/.virtualenvs/%(program_name)s/bin/gunicorn
  -w 4
  -b 127.0.0.1:7345
  --max-requests 2000
  --log-level debug
  --error-logfile=-
  --name %(program_name)s
  "application.app:create_app()"

environment=PATH="/home/root/.virtualenvs/%(program_name)s/bin"
numprocs=1
user=deploy
autostart=true
autorestart=true
redirect_stderr=true
redirect_stdout=true
stdout_logfile=/home/root/%(program_name)s-out.log
stdout_logfile_maxbytes=100MB
stdout_logfile_backups=10
stderr_logfile=/home/root/%(program_name)s-err.log
stderr_logfile_maxbytes=100MB
stderr_logfile_backups=10
```

supervisor 的常用命令如下：

```
supervisord -c supervisor.conf                                 通过配置文件启动 supervisor
supervisorctl -c supervisor.conf status                        查看 supervisor 的状态
supervisorctl -c supervisor.conf reload                        重新载入 配置文件
supervisorctl -c supervisor.conf start [all]|[appname]         启动指定/所有 supervisor 管理的程序进程
supervisorctl -c supervisor.conf stop [all]|[appname]          关闭指定/所有 supervisor 管理的程序进程
supervisorctl -c supervisor.conf restart [all]|[appname]       重启指定/所有 supervisor 管理的程序进程
```

使用 Nginx 做反向代理

将 Nginx 作为反向代理可以处理公共的 HTTP 请求，发送给 Gunicorn 并将响应带回给发送请求的客户端。在 ubuntu 上可以使用 `sudo apt-get install nginx` 安装 nginx，其他系统也类似。

要想配置 Nginx 作为运行在 127.0.0.1:7345 的 Gunicorn 的反向代理，我们可以在 /etc/nginx/sites-enabled 下给应用创建一个文件，不妨称之为 flask-todo-app.com，nginx 的类似配置如下：

```
# Handle requests to exploreflask.com on port 80
server {
    listen 80;
    server_name flask-todo-app.com;

    # Handle all locations
    location / {
        # Pass the request to Gunicorn
        proxy_pass http://127.0.0.1:7345;

        # Set some HTTP headers so that our app knows where the
request really came from
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_
for;
    }
}
```

常用的 nginx 使用命令如下：

```
$ sudo service nginx start
$ sudo service nginx stop
$ sudo service nginx restart
```

可以看到，我们上面的部署方式，都是手动部署的，如果有多台服务器要部署上面的程序，那就会是一个恶梦，有一个自动化部署的神器 Fabric 可以帮助我们解决这个问题，感兴趣的读者可以了解一下。

参考资料

- 部署 | Flask之旅
- python web 部署：nginx + gunicorn + supervisor + flask 部署笔记 - 简书
- 新手教程：建立网站的全套流程与详细解释 | 谢益辉

Flask 扩展插件

Flask 是一个简洁的 Web 框架，它提供了 Web 开发的核心功能，而丰富的扩展插件则让它变得更加强大。本章主要介绍 Flask 的扩展插件，主要有：

- Flask-Mail
- Flask-SQLAlchemy
- Flask-MongoEngine
- Flask-Cache
- Flask-HTTPAuth
- Flask-Bootstrap
- Flask-RESTful

Flask-Mail

给用户发送邮件是 Web 应用中最常见的任务之一，比如用户注册，找回密码等。Python 内置了一个 smtplib 的模块，可以用来发送邮件，这里我们使用 Flask-Mail，是因为它可以和 Flask 集成，让我们更方便地实现此功能。

安装

使用 `pip` 安装：

```
$ pip install Flask-Mail
```

或下载源码安装：

```
$ git clone https://github.com/mattupstate/flask-mail.git
$ cd flask-mail
$ python setup.py install
```

发送邮件

Flask-Mail 连接到简单邮件传输协议 (Simple Mail Transfer Protocol, SMTP) 服务器，并把邮件交给这个服务器发送。这里以 `QQ` 邮箱为例，介绍如何简单地发送邮件。在此之前，我们需要知道 `QQ` 邮箱的服务器地址和端口是什么，点此查看。

```python
# -*- coding: utf-8 -*-

from flask import Flask
from flask_mail import Mail, Message
import os

app = Flask(__name__)

app.config['MAIL_SERVER'] = 'smtp.qq.com'  # 邮件服务器地址
app.config['MAIL_PORT'] = 25               # 邮件服务器端口
app.config['MAIL_USE_TLS'] = True          # 启用 TLS
app.config['MAIL_USERNAME'] = os.environ.get('MAIL_USERNAME') or
'me@example.com'
app.config['MAIL_PASSWORD'] = os.environ.get('MAIL_PASSWORD') or
'123456'

mail = Mail(app)

@app.route('/')
def index():
    msg = Message('Hi', sender='me@example.com', recipients=['he
@example.com'])
    msg.html = '<b>Hello Web</b>'
    # msg.body = 'The first email!'
    mail.send(msg)
    return '<h1>OK!</h1>'

if __name__ == '__main__':
    app.run(host='127.0.0.1', debug=True)
```

在发送前，需要先设置用户名和密码，当然你也可以直接写在文件里，如果是从环境变量读取，可以这么做：

```
$ export MAIL_USERNAME='me@example.com'
$ export MAIL_PASSWORD='123456'
```

将上面的 `sender` 和 `recipients` 改一下，就可以进行测试了。

从上面的代码，我们可以知道，使用 Flask-Mail 发送邮件主要有以下几个步骤：

- 配置 app 对象的邮件服务器地址，端口，用户名和密码等
- 创建一个 Mail 的实例： `mail = Mail(app)`
- 创建一个 Message 消息实例，有三个参数：邮件标题、发送者和接收者
- 创建邮件内容，如果是 HTML 格式，则使用 `msg.html` ，如果是纯文本格式，则使用 `msg.body`
- 最后调用 `mail.send(msg)` 发送消息

Flask-Mail 配置项

Flask-Mail 使用标准的 Flask 配置 API 进行配置，下面是一些常用的配置项：

配置项	说明
MAIL_SERVER	邮件服务器地址，默认为 localhost
MAIL_PORT	邮件服务器端口，默认为 25
MAIL_USE_TLS	是否启用传输层安全 (Transport Layer Security, TLS)协议，默认为 False
MAIL_USE_SSL	是否启用安全套接层 (Secure Sockets Layer, SSL)协议，默认为 False
MAIL_DEBUG	是否开启 DEBUG，默认为 app.debug
MAIL_USERNAME	邮件服务器用户名，默认为 None
MAIL_PASSWORD	邮件服务器密码，默认为 None
MAIL_DEFAULT_SENDER	邮件发件人，默认为 None，也可在 Message 对象里指定
MAIL_MAX_EMAILS	邮件批量发送个数上限，默认为 None
MAIL_SUPPRESS_SEND	默认为 app.testing，如果为 True，则不会真的发送邮件，供测试用

异步发送邮件

使用上面的方式发送邮件，会发现页面卡顿了几秒才出现消息，这是因为我们使用了同步的方式。为了避免发送邮件过程中出现的延迟，我们把发送邮件的任务移到后台线程中，代码如下：

```
# -*- coding: utf-8 -*-

from flask import Flask
from flask_mail import Mail, Message
from threading import Thread
import os

app = Flask(__name__)

app.config['MAIL_SERVER'] = 'smtp.qq.com'
app.config['MAIL_PORT'] = 25
app.config['MAIL_USE_TLS'] = True
app.config['MAIL_USERNAME'] = os.environ.get('MAIL_USERNAME') or
'smtp.example.com'
app.config['MAIL_PASSWORD'] = os.environ.get('MAIL_PASSWORD') or
'123456'

mail = Mail(app)

def send_async_email(app, msg):
    with app.app_context():
        mail.send(msg)

@app.route('/sync')
def send_email():
    msg = Message('Hi', sender='me@example.com', recipients=['he
@example.com'])
    msg.html = '<b>send email asynchronously</b>'

    thr = Thread(target=send_async_email, args=[app, msg])
    thr.start()
    return 'send successfully'

if __name__ == '__main__':
    app.run(host='127.0.0.1', debug=True)
```

在上面，我们创建了一个线程，执行的任务是 `send_async_email` ，该任务的实现涉及一个问题[1]：

> 很多 Flask 扩展都假设已经存在激活的程序上下文和请求上下文。Flask-Mail 中的 `send()` 函数使用 `current_app`，因此必须激活程序上下文。不过，在不同线程中执行 `mail.send()` 函数时，程序上下文要使用 `app.app_context()` 人工创建。

带附件的邮件

有时候，我们发邮件的时候需要添加附件，比如文档和图片等，这也很简单，代码如下：

```
# -*- coding: utf-8 -*-

from flask import Flask
from flask_mail import Mail, Message
import os

app = Flask(__name__)

app.config['MAIL_SERVER'] = 'smtp.qq.com'  # 邮件服务器地址
app.config['MAIL_PORT'] = 25               # 邮件服务器端口
app.config['MAIL_USE_TLS'] = True          # 启用 TLS
app.config['MAIL_USERNAME'] = os.environ.get('MAIL_USERNAME') or
'me@example.com'
app.config['MAIL_PASSWORD'] = os.environ.get('MAIL_PASSWORD') or
'123456'

mail = Mail(app)

@app.route('/attach')
def add_attchments():
    msg = Message('Hi', sender='me@example.com', recipients=['ot
her@example.com'])
    msg.html = '<b>Hello Web</b>'

    with app.open_resource("/Users/Admin/Documents/pixel-example
.jpg") as fp:
        msg.attach("photo.jpg", "image/jpeg", fp.read())

    mail.send(msg)
    return '<h1>OK!</h1>'

if __name__ == '__main__':
    app.run(host='127.0.0.1', debug=True)
```

上面的代码中，我们通过 `app.open_resource(path_of_attachment)` 打开了本机的某张图片，然后通过 `msg.attach()` 方法将附件内容添加到 Message 对象。 `msg.attach()` 方法的第一个参数是附件的文件名，第二个参数是文件内容的 `MIME (Multipurpose Internet Mail Extensions)` 类型，第三个参数是文件内容。

如果你不知道附件的 `MIME` 类型是什么，可以查看 MIME 参考手册。

批量发送

在某些情况下，我们需要批量发送邮件，比如给网站的所有注册用户发送改密码的邮件，这时为了避免每次发邮件时都要创建和关闭跟服务器的连接，我们的代码需要做一些调整，类似如下：

```
with mail.connect() as conn:
    for user in users:
        subject = "hello, %s" % user.name
        msg = Message(recipients=[user.email], body='...', subje
ct=subject)
        conn.send(msg)
```

上面的工作方式，使得应用与电子邮件服务器保持连接，一直到所有邮件已经发送完毕。某些邮件服务器会限制一次连接中的发送邮件的上限，这样的话，你可以配置 `MAIL_MAX_EMAILS` 。

需要注意的是，更好的发送大量电子邮件的方式是用专门的作业系统，比如用 Celery 任务队列等。

本文完整的代码在这里。

更多阅读

- flask-mail — Flask-Mail 0.9.1 documentation
- Flask扩展系列(二)–Mail – 思诚之道

> 1. Flask Web Development ↩

Flask-SQLAlchemy

ORM 框架

Web 开发中，一个重要的组成部分便是数据库了。Web 程序中最常用的莫过于关系型数据库了，也称 SQL 数据库。另外，文档数据库（如 mongodb）、键值对数据库（如 redis）近几年也逐渐在 web 开发中流行起来，我们习惯把这两种数据库称为 NoSQL 数据库。

大多数的关系型数据库引擎（比如 MySQL、Postgres 和 SQLite）都有对应的 Python 包。在这里，我们不直接使用这些数据库引擎提供的 Python 包，而是使用对象关系映射（Object-Relational Mapper, ORM）框架，它将低层的数据库操作指令抽象成高层的面向对象操作。也就是说，如果我们直接使用数据库引擎，我们就要写 SQL 操作语句，但是，如果我们使用了 ORM 框架，我们对诸如表、文档此类的数据库实体就可以简化成对 Python 对象的操作。

Python 中最广泛使用的 ORM 框架是 SQLAlchemy，它是一个很强大的关系型数据库框架，不仅支持高层的 ORM，也支持使用低层的 SQL 操作，另外，它也支持多种数据库引擎，如 MySQL、Postgres 和 SQLite 等。

Flask-SQLAlchemy

在 Flask 中，为了简化配置和操作，我们使用的 ORM 框架是 Flask-SQLAlchemy，这个 Flask 扩展封装了 SQLAlchemy 框架。在 Flask-SQLAlchemy 中，数据库使用 URL 指定，下表列出了常见的数据库引擎和对应的 URL。

数据库引擎	URL
MySQL	mysql://username:password@hostname/database
Postgres	postgresql://username:password@hostname/database
SQLite (Unix)	sqlite:////absolute/path/to/database
SQLite (Windows)	sqlite:///c:/absolute/path/to/database

上面的表格中，username 和 password 表示登录数据库的用户名和密码，hostname 表示 SQL 服务所在的主机，可以是本地主机（localhost）也可以是远程服务器，database 表示要使用的数据库。有一点需要注意的是，SQLite 数据库不需要使用服务器，它使用硬盘上的文件名作为 database。

一个最小的应用

创建数据库

首先，我们使用 pip 安装 Flask-SQLAlchemy:

```
$ pip install flask-sqlalchemy
```

接下来，我们配置一个简单的 SQLite 数据库：

```
$ cat app.py
# -*- coding: utf-8 -*-

from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///db/users.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = True
db = SQLAlchemy(app)


class User(db.Model):
    """定义数据模型"""
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True)
    email = db.Column(db.String(120), unique=True)

    def __init__(self, username, email):
        self.username = username
        self.email = email

    def __repr__(self):
        return '<User %r>' % self.username
```

这里有几点需要注意的是：

1. app 应用配置项 `SQLALCHEMY_DATABASE_URI` 指定了 SQLAlchemy 所要操作的数据库，这里我们使用的是 SQLite，数据库 URL 以 `sqlite:///` 开头，

后面的 `db/users.db` 表示数据库文件存放在当前目录的 `db` 子目录中的 `users.db` 文件。当然，你也可以使用绝对路径，如 `/tmp/users.db` 等。
2. db 对象是 SQLAlchemy 类的实例，表示程序使用的数据库。
3. 我们定义的 `User` 模型必须继承自 `db.Model`，这里的模型其实就对应着数据库中的表。其中，类变量 `__tablename__` 定义了在数据库中使用的表名，如果该变量没有被定义，Flask-SQLAlchemy 会使用一个默认名字。

接着，我们创建表和数据库。为此，我们先在当前目录创建 `db` 子目录和新建一个 `users.db` 文件，然后在交互式 Python shell 中导入 db 对象并调用 SQLAlchemy 类的 create_all() 方法：

```
$ mkdir db
$ python
>>> from app import db
>>> db.create_all()
```

我们验证一下，"users" 表是否创建成功：

```
$ sqlite3 db/users.db    # 打开数据库文件
SQLite version 3.8.10.2 2015-05-20 18:17:19
Enter ".help" for usage hints.

sqlite> .schema users   # 查看 "user" 表的 schema
CREATE TABLE users (
        id INTEGER NOT NULL,
        username VARCHAR(80),
        email VARCHAR(120),
        PRIMARY KEY (id),
        UNIQUE (username),
        UNIQUE (email)
);
```

插入数据

现在，我们创建一些用户，通过使用 `db.session.add()` 来添加数据：

```
@app.route('/adduser')
def add_user():
    user1 = User('ethan', 'ethan@example.com')
    user2 = User('admin', 'admin@example.com')
    user3 = User('guest', 'guest@example.com')
    user4 = User('joe', 'joe@example.com')
    user5 = User('michael', 'michael@example.com')

    db.session.add(user1)
    db.session.add(user2)
    db.session.add(user3)
    db.session.add(user4)
    db.session.add(user5)

    db.session.commit()

    return "<p>add succssfully!"
```

这里有一点要注意的是，我们在将数据添加到会话后，在最后要记得调用 `db.session.commit()` 提交事务，这样，数据才会被写入到数据库。

查询数据

查询数据主要是用 `query` 接口，例如 `all()` 方法返回所有数据，`filter_by()` 方法对查询结果进行过滤，参数是键值对，`filter` 方法也可以对结果进行过滤，但参数是布尔表达式，详细的介绍请查看这里。

```
>>> from app import User
>>> users = User.query.all()
>>> users
[<User u'ethan'>, <User u'admin'>, <User u'guest'>, <User u'joe'
>, <User u'michael'>]
>>>
>>> user = User.query.filter_by(username='joe').first()
>>> user
<User u'joe'>
>>> user.email
u'joe@example.com'
>>>
>>> user = User.query.filter(User.username=='ethan').first()
>>> user
<User u'ethan'>
```

如果我们想查看 SQLAlchemy 为查询生成的原生 SQL 语句，只需要把 query 对象转化成字符串：

```
>>> str(User.query.filter_by(username='guest'))
'SELECT users.id AS users_id, users.username AS users_username,
users.email AS users_email \nFROM users \nWHERE users.username =
 :username_1'
```

分页方法

分页方法可以采用 `limit()` 和 `offset()` 方法，比如从第 3 条记录开始取(注意是从 0 开开始算起)，并最多取 1 条记录，可以这样：

```
users = User.query.limit(1).offset(3)
```

更新数据

更新数据也用 `add()` 方法，如果存在要更新的对象，SQLAlchemy 就更新该对象而不是添加。

```
>>> from app import db
>>> from app import User
>>>
>>> admin = User.query.filter_by(username='admin').first()
>>>
>>> admin.email = 'admin@hotmail.com'
>>> db.session.add(admin)
>>> db.session.commit()
>>>
>>> admin = User.query.filter_by(username='admin').first()
>>> admin.email
u'admin@hotmail.com'
```

删除数据

删除数据用 `delete()` 方法，同样要记得 `delete` 数据后，要调用 `commit()` 提交事务：

```
>>> from app import db
>>> from app import User
>>>
>>> admin = User.query.filter_by(username='admin').first()
>>> db.session.delete(admin)
>>> db.session.commit()
```

本文完整的代码可在这里下载。

更多阅读

- Flask-SQLAlchemy

Flask-MongoEngine

MongoDB 是一个文档型数据库，是 NoSQL (not only SQL) 的一种，具有灵活、易扩展等诸多优点，受到许多开发者的青睐。MongoEngine 是一个用来操作 MongoDB 的 ORM 框架，如果你不知道什么是 ORM，可以参考 Flask-SQLAlchemy 一节。在 Flask 中，我们可以直接使用 MongoEngine，也可使用 Flask-MongoEngine ，它使得在 Flask 中使用 MongoEngine 变得更加简单。

安装

使用 pip 安装，如下：

```
$ pip install flask-mongoengine
```

使用

在使用之前，请确保 mongo 服务已经开启。

配置

我们需要配置 MongoDB 的相关参数，以便我们能访问数据库。

```
# -*- coding: utf-8 -*-

from flask import Flask
from flask_mongoengine import MongoEngine

app = Flask(__name__)
app.config['MONGODB_SETTINGS'] = {
    'db': 'test',
    'host': '127.0.0.1',
    'port': 27017
}

db = MongoEngine(app)
```

上面的代码中，我们在 `app.config` 的 `MONGODB_SETTINGS` 字典中配置了数据库、主机和端口。如果数据库需要身份验证，那我们需要在该字典中添加 `username` 和 `password` 参数，比如：

```
app.config['MONGODB_SETTINGS'] = {
  'db': 'test',
  'username':'admin',
  'password':'12345'
}
```

另外，上面的配置也可以改成下面的方式：

```
app.config['MONGODB_DB'] = 'test'
app.config['MONGODB_HOST'] = '127.0.0.1'
app.config['MONGODB_PORT'] = 27017
app.config['MONGODB_USERNAME'] = 'admin'
app.config['MONGODB_PASSWORD'] = '12345'
```

如果我们想在应用初始化前配置数据库，比如使用工厂方法，可以类似这样做：

```
from flask import Flask
from flask_mongoengine import MongoEngine
db = MongoEngine()

...

app = Flask(__name__)
app.config.from_pyfile('config.json')
db.init_app(app)
```

定义数据模型

接下来，我们需要定义数据模型。这里，我们以一个 `Todo` 数据库为例，数据模型定义如下：

```
from datetime import datetime

class Todo(db.Document):
    meta = {
        'collection': 'todo',
        'ordering': ['-create_at'],
        'strict': False,
    }

    task = db.StringField()
    create_at = db.DateTimeField(default=datetime.now)
    is_completed = db.BooleanField(default=False)
```

在上面的代码中，我们定义了一个 `Todo` 类， `meta` 字典设置了 `collection` ， `ordering` 和 `strict` ，其中 `ordering` 的值可以指定你的 QuerySet 的默认顺序， `strict` 的值指定是否使用严格模式，默认值是 `True` ，也就是使用严格模式，这就意味着如果数据库的记录如果存在某些字段没有在我们的数据模型中声明，那程序在运行时会产生一个 `FieldDoesNotExist` 的错误。因此，我们的数据模型定义最好跟记录中的字段保持一致。

查询数据

- 查询所有数据使用 `all()` 方法

```
todos = Todo.objects().all()
```

- 查询满足某些条件的数据

```
task = 'cooking'
todo = Todo.objects(task=task).first()
```

其中， `first()` 方法会取出满足条件的第 1 条记录。

添加数据

- 添加数据使用 `save()` 方法

```
todo1 = Todo(task='task 1', is_completed=False)
todo1.save()
```

数据排序

- 排序使用 `order_by()` 方法

```
todos = Todo.objects().order_by('create_at')
```

更新数据

- 更新数据需要先查找数据，然后再更新

```
task = 'task 1'
todo = Todo.objects(task=task).first()  # 先查找
if not todo:
    return "the task doesn't exist!"

todo.update(is_completed=True)   # 再更新
```

删除数据

- 删除数据使用 `delete()` 方法：先查找，再删除

```
task = 'task 6'
todo = Todo.objects(task=task).first()  # 先查找
if not todo:
    return "the task doesn't exist!"

todo.delete()   # 再删除
```

分页

- 分页可结合使用 `skip()` 和 `limit()` 方法

```
skip_nums = 1
limit = 3

todos = Todo.objects().order_by(
    '-create_at'
).skip(
    skip_nums
).limit(
    limit
)
```

- 使用 `paginate()` 方法

```
def view_todos(page=1):
    todos = Todo.objects.paginate(page=page, per_page=10)
```

本文完整的代码在这里。

更多阅读

- flask-mongoengine

Flask-Cache

假设你的 Web 服务对于某些请求比较耗时，而该请求的返回结果在较短的时间内（比如 5 分钟内）都是足够有效的，这时你能想到什么方法去改善这种状况呢？缓存？对，至少这是一种提高性能的最简单的方法。

Flask 本身不提供缓存功能，但是作为 Flask 核心的 Werkzeug 框架则提供了一个简单的缓存对象 SimpleCache，它将缓存项存放在 Python 解释器的内存中。使用 `SimpleCache` 需要自己实现缓存装饰器，稍微有点麻烦。幸运的是，Flask-Cache 扩展使我们在 Flask 中使用缓存变得很简单。

安装

使用 pip 安装：

```
$ pip install Flask-Cache
```

使用

缓存视图函数

同使用其他扩展一样，我们要先创建扩展的实例：

```
from flask import Flask
from flask_cache import Cache

app = Flask(__name__)
cache = Cache(app, config={'CACHE_TYPE': 'simple'})
```

同样，我们也可以初始化 `Cache` 后再使用 `init_app` 方法：

```
from flask import Flask
from flask_cache import Cache

cache = Cache(config={'CACHE_TYPE': 'simple'})

app = Flask(__name__)
cache.init_app(app)
```

上面的代码中，我们设置 `CACHE_TYPE` 即缓存类型为 'simple'，其内部实现就是 Werkzeug 中的SimpleCache。

另外，我们也可以使用第三方的缓存服务器，比如 Redis，可以这样设置：

```
cache = Cache(
    app,
    config={
        'CACHE_TYPE': 'redis',
        'CACHE_REDIS_HOST': '127.0.0.1',
        'CACHE_REDIS_PORT': 6379,
        'CACHE_REDIS_PASSWORD': '123456',
        'CACHE_REDIS_DB': 0
    }
)
```

现在，让我们来看一个简单的例子，代码如下：

```
# -*- coding: utf-8 -*-

from flask import Flask
from flask_cache import Cache

app = Flask(__name__)
cache = Cache(app, config={'CACHE_TYPE': 'simple'})

@app.route('/')
@cache.cached(timeout=300)
def index():
    print 'view hello called'
    return 'Hello World!'

if __name__ == '__main__':
    app.run(host='127.0.0.1', port=5205, debug=True)
```

上面的代码中，我们使用了 `cache` 对象的 `cached()` 方法来装饰视图函数，在 `cached()` 方法中，我们设置了 `timeout` 参数为 300 (单位为：秒)，这就意味着该视图函数在每 5 分钟最多运行一次，响应的结果会被保存在缓存中，并可以让期间的每一个请求获取。

那我们怎么验证上面的视图函数是每 5 分钟最多运行一次呢？其实很简单，我们把上面的代码跑起来，浏览器访问 `http://127.0.0.1:5205/` ，这时出现 Hello World!，在终端可以看到 view hello called，刷新页面，也可以看到 Hello World!，但是终端已经看不到 view hello called 了，过 5 分钟后，重复上面的步骤，这时在终端又可以看到 view hello called 了，这说明什么呢？缓存起效了！

在终端的打印日志如下：

```
$ python app.py
 * Running on http://127.0.0.1:5205/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 109-566-036
view hello called
127.0.0.1 - - [20/Oct/2016 22:50:28] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Oct/2016 22:50:29] "GET /favicon.ico HTTP/1.1"
 404 -
127.0.0.1 - - [20/Oct/2016 22:50:42] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Oct/2016 22:52:16] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Oct/2016 22:52:59] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Oct/2016 22:53:14] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [20/Oct/2016 22:55:27] "GET / HTTP/1.1" 200 -
view hello called
127.0.0.1 - - [20/Oct/2016 22:55:37] "GET / HTTP/1.1" 200 -
```

缓存普通函数

上面，我们对视图函数进行了缓存。事实上，我们也可以给非视图函数加缓存，比如：

```
@app.route('/posts')
def get_posts():
    return ', '. join(get_posts())

# 对非视图函数进行缓存
@cache.cached(timeout=60, key_prefix='get_posts')
def get_posts():
    print 'method get_posts called'
    return ['a', 'b', 'c', 'd']
```

给非视图函数加缓存，跟缓存视图函数类似，但有一点需要注意的是，此时 `cached()` 方法需要加上 `key_prefix` 参数，由于视图函数在默认情况下使用请求路径 (request.path) 作为 cache_key，所以可以省略。

运行上面的代码，在终端可以看到 method get_posts called，刷新页面，只有过了 60 秒后，在终端才能再次看到。

使用 Memoization（一种缓存技术）

上面的普通函数是没有带参数的，如果带参数，就不能像上面那样做了，为什么？不妨一试：

```
@app.route('/comments/<int:num>')
def get_comments(num):
    return ', '.join(get_comments(num))

# 缓存带参数的函数
@cache.cached(timeout=60, key_prefix='get_comments')
def get_comments(num):
    print 'method get_comments called'
    return [str(i) for i in xrange(num)]
```

运行上面的代码，会发生什么情况呢？

我们发现：当在浏览器输入 `http://127.0.0.1:5205/comments/3` 时，返回了 '0, 1, 2'，同时在终端看到 'method get_comments called'，这是符合预期的。但是，如果此时我们把链接改成 `http://127.0.0.1:5205/comments/4` ，这时依然返回 '0, 1, 2'，而我们想要的应该是 '0, 1, 2, 3'。仔细思考下，不能发现原因，因为我们对函数的结果进行了缓存，这时不管输入是什么，输出都是一样的，只有过了 60 秒后，结果才会改变。

那应该怎么改进呢？其实很简单，我们不用 `cache.cached` ，而用 Flask-Cache 提供的另一个装饰器方法 `cache.memoize` ，它与 `cache.cached` 的区别就是它会将函数的参数也放在缓存项的键值中。

我们将上面的代码改写如下：

```
@app.route('/comments/<int:num>')
def get_comments(num):
    return ', '.join(get_comments(num))

# 缓存带参数的函数
@cache.memoize(timeout=60)
def get_comments(num):
    print 'method get_comments called'
    return [str(i) for i in xrange(num)]
```

运行上面的代码，在浏览器输入 `http://127.0.0.1:5205/comments/3` ，返回 '0, 1, 2'，同时终端打印出 'method_get_comments called'，刷新页面，返回没有改变，终端也没有打印出上述字样，这时把链接改成 `http://127.0.0.1:5205/comments/4` ，返回了 '0, 1, 2, 3'，同时终端也打印出了上述字样，符合我们的预期。

小结

> 如果函数不接受参数的话，cached() 和 memoize() 两者的作用是一样的。

删除缓存

- 使用 `delete()` 方法删除普通缓存

```
cache.delete('get_posts')
```

- 使用 `delete_many()` 方法删除多个缓存

```
cache.delete_many('get_posts', 'index')
```

- 使用 `delete_memoized()` 方法删除 memoize 缓存

```
# 删除调用'get_comments'函数并且参数为 3 的缓存项
cache.delete_memoized('get_comments')
```

- 使用 `clear()` 方法删除所有缓存

```
cache.clear()
```

本文完整的代码在这里。

更多阅读

- Flask-Cache
- Flask扩展系列(六)–缓存 – 思诚之道

Flask-HTTPAuth

在 Web 应用中，我们经常需要保护我们的 api，以避免非法访问。比如，只允许登录成功的用户发表评论等。Flask-HTTPAuth 扩展可以很好地对 HTTP 的请求进行认证，不依赖于 Cookie 和 Session。本文主要介绍两种认证的方式：基于密码和基于令牌 (token)。

安装

使用 pip 安装：

```
$ pip install Flask-HTTPAuth
```

基于密码的认证

为了简化代码，这里我们就不引入数据库了。

- 首先，创建扩展对象实例

```
from flask import Flask
from flask_httpauth import HTTPBasicAuth

app = Flask(__name__)
auth = HTTPBasicAuth()
```

这里有一点需要注意的是，我们创建了一个 `auth` 对象，但没有传入 `app` 对象，这跟其他扩展初始化实例有一点区别。

- 接着，写一个验证用户密码的回调函数

```
from werkzeug.security import generate_password_hash, check_password_hash

# 模拟数据库
books = ['The Name of the Rose', 'The Historian', 'Rebecca']
users = [
    {'username': 'ethan', 'password': generate_password_hash('6666')},
    {'username': 'peter', 'password': generate_password_hash('4567')}
]

# 回调函数
@auth.verify_password
def verify_password(username, password):
    user = filter(lambda user: user['username'] == username, users)

    if user and check_password_hash(user[0]['password'], password):
        g.user = username
        return True
    return False
```

上面，为了对密码进行加密以及认证，我们使用 `werkzeug.security` 包提供的 `generate_password_hash` 和 `check_password_hash` 方法：`generate_password_hash` 会对给定的字符串，生成其加盐的哈希值；`check_password_hash` 验证传入的明文字符串与哈希值是否一致。

- 然后，我们在需要认证的视图函数上，加上 `@auth.login_required` 装饰器，比如

```
@app.route('/', methods=['POST'])
@auth.login_required
def add_book():
    _form = request.form
    title = _form["title"]
    if not title:
        return '<h1>invalid request</h1>'

    books.append(title)
    flash("add book successfully!")
    return redirect(url_for('index'))
```

上面完整的代码如下：

```
$ cat app.py

# -*- coding: utf-8 -*-

from flask import Flask, url_for, render_template, request, flas
h, \
    redirect, make_response, jsonify, g
from werkzeug.security import generate_password_hash, check_pass
word_hash
from flask_httpauth import HTTPBasicAuth


app = Flask(__name__)
app.config['SECRET_KEY'] = 'secret key'

auth = HTTPBasicAuth()

# 模拟数据库
books = ['The Name of the Rose', 'The Historian', 'Rebecca']
users = [
    {'username': 'ethan', 'password': generate_password_hash('66
66')},
    {'username': 'peter', 'password': generate_password_hash('45
67')}
]

# 回调函数
```

```
@auth.verify_password
def verify_password(username, password):
    user = filter(lambda user: user['username'] == username, use
rs)

    if user and check_password_hash(user[0]['password'], passwor
d):
        g.user = username
        return True
    return False

# 不需认证，可直接访问
@app.route('/', methods=['GET'])
def index():
    return render_template(
        'book.html',
        books=books
    )

# 需要认证
@app.route('/', methods=['POST'])
@auth.login_required
def add_book():
    _form = request.form
    title = _form["title"]
    if not title:
        return '<h1>invalid request</h1>'

    books.append(title)
    flash("add book successfully!")
    return redirect(url_for('index'))

@auth.error_handler
def unauthorized():
    return make_response(jsonify({'error': 'Unauthorized access'
}), 401)

if __name__ == '__main__':
    app.run(host='127.0.0.1', port=5206, debug=True)
```

```
$ cat templates/layout.html

<!doctype html>
<title>Hello Sample</title>

<div class="page">
    {% block body %} {% endblock %}
</div>

{% for message in get_flashed_messages() %}
    {{ message }}
{% endfor %}
```

```
$ cat templates/book.html

{% extends "layout.html" %}
{% block body %}
{% if books %}
    {% for book in books %}
        <ul>
            <li> {{ book }} </li>
        </ul>
    {% endfor %}
{% else %}
    <p> The book doesn't exists! </p>
{% endif %}

<form method="post" action="{{ url_for('add_book') }}">
    <input id="title" name="title" placeholder="add book" type="
text">
    <button type="submit">Submit</button>
</form>

{% endblock %}
```

保存上面的代码，启动该应用，当我们试图提交书籍时，你会浏览器弹出了一个登录框，如下，只有输入正确的用户名和密码，书籍才会添加成功。

需要进行身份验证
http://127.0.0.1:5206 要求提供用户名和密码。
用户名：
密码：
取消
登录

基于 token 的认证

很多时候，我们并不直接通过密码做认证，比如当我们把 api 开放给第三方的时候，我们不可能给它们提供密码，而是对它们进行授权，还可能会有时间限制，比如半年或一年等。这时候，我们往往通过一个令牌，也就是 token 来做认证，Flask-HTTPAuth 提供了 HTTPTokenAuth 对象来做这件事。

我们用官方文档的例子来说明。

```
from flask import Flask, g
from flask_httpauth import HTTPTokenAuth

app = Flask(__name__)
auth = HTTPTokenAuth(scheme='Token')

tokens = {
    "secret-token-1": "john",
    "secret-token-2": "susan"
}

# 回调函数，验证 token 是否合法
@auth.verify_token
def verify_token(token):
    if token in tokens:
        g.current_user = tokens[token]
        return True
    return False

# 需要认证
@app.route('/')
@auth.login_required
def index():
    return "Hello, %s!" % g.current_user

if __name__ == '__main__':
    app.run()
```

上面，我们在初始化 HTTPTokenAuth 对象时，传入了 `scheme='Token'` 。这个 scheme，是我们在发送请求时，在 HTTP 头 `Authorization` 中要用的 scheme 字段。用 curl 测试如下：

```
$ curl -X GET -H "Authorization: Token secret-token-1" http://localhost:5000/
```

结果：

```
Hello, john!
```

使用 itsdangerous 库来管理令牌

上面的令牌还是比较薄弱的，在实际使用中，我们需要使用加密的签名（Signature）作为令牌，它能够根据用户信息生成相关的签名，并且很难被篡改。itsdangerous 提供了上述功能，在使用之前请使用 pip 安装：`$ pip install itsdangerous` 。

改进后的代码如下：

```
# -*- coding: utf-8 -*-

from flask import Flask, g
from flask_httpauth import HTTPTokenAuth
from itsdangerous import TimedJSONWebSignatureSerializer as Seri
alizer

app = Flask(__name__)
app.config['SECRET_KEY'] = 'secret key here'

auth = HTTPTokenAuth(scheme='Token')

# 实例化一个签名序列化对象 serializer，有效期 10 分钟
serializer = Serializer(app.config['SECRET_KEY'], expires_in=600
)

users = ['john', 'susan']

# 生成 token
for user in users:
    token = serializer.dumps({'username': user})
    print('Token for {}: {}\n'.format(user, token))

# 回调函数，对 token 进行验证
@auth.verify_token
def verify_token(token):
    g.user = None
    try:
        data = serializer.loads(token)
    except:
        return False
    if 'username' in data:
```

```
        g.user = data['username']
        return True
    return False

# 对视图进行认证
@app.route('/')
@auth.login_required
def index():
    return "Hello, %s!" % g.user

if __name__ == '__main__':
    app.run()
```

将上面代码保存为 `app.py` ，在终端运行，可看到类似如下的输出：

```
$ python app.py
Token for John: eyJhbGciOiJIUzI1NiIsImV4cCI6MTQ3NjY5NzE0NCwiaWF0
IjoxNDc2Njk1MzQ0fQ.eyJ1c2VybmFtZSI6IkpvaG4ifQ.vQu0z0Pos2Tgt5jBYM
Y5IYWUkTK9k3wE_RqvYHDqtyM

Token for Susan: eyJhbGciOiJIUzI1NiIsImV4cCI6MTQ3NjY5NzE0NCwiaWF
0IjoxNDc2Njk1MzQ0fQ.eyJ1c2VybmFtZSI6IlN1c2FuIn0.rk8JaTRwag0qiF9_
KuRodhw6wx2ZWkOEhFln9hzOLP0

 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

使用 curl 测试如下：

```
$ curl -X GET -H "Authorization: Token eyJhbGciOiJIUzI1NiIsImV4c
CI6MTQ3NjY5NzE0NCwiaWF0IjoxNDc2Njk1MzQ0fQ.eyJ1c2VybmFtZSI6IkpvaG
4ifQ.vQu0z0Pos2Tgt5jBYMY5IYWUkTK9k3wE_RqvYHDqtyM" http://localho
st:5000/

# 结果
$ Hello, john!
```

本节的代码可在这里下载。

更多阅读

- 使用 Flask 设计 RESTful 的认证 — Designing a RESTful API with Python and Flask 1.0 文档
- Flask扩展系列(九)–HTTP认证 – 思诚之道
- miguelgrinberg/REST-auth: Example application for my RESTful Authentication with Flask article.
- Welcome to Flask-HTTPAuth's documentation! — Flask-HTTPAuth documentation

Flask-Bootstrap

Bootstrap 是 Twitter 开源的一个 CSS/HTML 框架，它让 Web 开发变得更加迅速，简单。要想在我们的 Flask 应用中使用 Boostrap，有两种方案可供选择：

- 第 1 种，在我们的 Jinja 模板中直接引入 Bootstrap 层叠样式表 (CSS) 和 JavaScript 文件，比如 bootstrap.min.css，bootstrap.min.js；
- 第 2 种，也是更简单的方法，就是使用一个 Flask-Bootstrap 的扩展，它简化了集成 Bootstrap 的过程；

安装

使用 `pip` 安装：

```
$ pip install flask-bootstrap
```

使用

现在，让我们写一个简单的 `hello world` 例子。

首先，新建一个 `app.py` 文件，代码如下：

```
# -*- coding: utf-8 -*-

from flask import Flask, render_template
from flask_bootstrap import Bootstrap

app = Flask(__name__)
Bootstrap(app)  # 把程序实例即 app 传入构造方法进行初始化

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/book')
def book():
    books = ['the first book', 'the second book']
    return render_template('index.html', books=books)

@app.route('/movie')
def movie():
    movies = ['the first movie', 'the second movie']
    return render_template('index.html', movies=movies)

if __name__ == '__main__':
    app.run(host='127.0.0.1', port=5200, debug=True)
```

初始化 Flask-Bootstrap 之后，我们的模板通过继承一个基模板，即 `bootstrap/base.html` ，就可以使用 Bootstrap 了。

新建一个 `templates` 文件夹，在该文件夹中添加一个 `index.html` 的文件，代码如下：

```
{% extends "bootstrap/base.html" %}

{% block title %}Demo{% endblock %}

{% block navbar %}
<div class="navbar navbar-inverse">
    <div class="container">
        <div class="navbar-header">
            <a class="navbar-brand" href="{{ url_for('index') }}
">首页</a>
```

```
        </div>
        <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
                <li><a href="{{ url_for('book') }}">读书</a></li>
                <li><a href="{{ url_for('movie') }}">电影</a></li
>
            </ul>
        </div>
    </div>
</div>
{% endblock %}

{% block content %}
<div class="container">
    {% if books %}
        <ul>
            {% for book in books %}
                <li>{{ book }}</li>
            {% endfor %}
        </ul>
    {% elif movies %}
        <ul>
            {% for movie in movies %}
                <li>{{ movie }}</li>
            {% endfor %}
        </ul>
    {% else %}
        <p><a href="{{ url_for('book') }}"> 读书 </a></p>
        <p><a href="{{ url_for('movie') }}"> 电影 </a></p>
    {% endif %}
</div>
{% endblock %}
```

注意到上面模板中的第 1 行：

```
{% extends "bootstrap/base.html" %}
```

这是利用了 Jinja2 的模板继承机制，我们在前面也已经讲过了，如果忘了，可点此查看。 `extends` 指令从 Flask-Bootstrap 中导入 `bootstrap/base.html` ，该基模板引入了 Bootstrap 中的所有 CSS 和 JavaScript 文件。通过这一句，我们就可以在模板中使用 Bootstrap 了，是不是很简单？

注意到，我们在模板中也定义了诸如 `{% block title %} Demo {% endblock %}` 的块，这些块是基模板提供的，可在我们的衍生模板中重新定义。其中， `title` 块中的内容会被放到渲染后的 `<title>` 标签中， `navbar` 块表示页面中的导航条， `content` 块表示页面的主体内容。

基模板除了提供上面的块，还定义了很多其他块，下面的表格列出了所有可用的块：

块名	说明
doc	整个 HTML 文档
html	`<html>` 标签中的内容
html_attribs	`<html>` 标签的属性
head	`<head>` 标签中的内容
body	`<body>` 标签中的内容
body_attribs	`<body>` 标签的属性
title	`<title>` 标签中的内容
styles	CSS 定义
metas	一组 `<meta>` 标签
navbar	导航条
content	页面内容
scripts	文档底部的 JavaScript 声明

上面的块，比如 `title` , `navbar` 和 `content` 是可以直接重定义的，但是，如果程序需要向已经有内容的块添加新内容，必须使用 Jinja2 提供的 `super()` 函数，比如 `styles` 块和 `scripts` 块：

- 如果我们想添加自己的样式表，需要这样：

```
{% block styles %}
{{ super() }}
<link rel="stylesheet" href="{{url_for('.static', filename='myst
yle.css')}}">
{% endblock %}
```

- 如果我们想添加自己的 JavaScript 文件，需要这样：

```
{% block scripts %}
{{ super() }}
<script src="{{url_for('.static', filename='myscripts.js')}}"></
script>
{% endblock %}
```

本文完整的代码可在这里下载。

更多阅读

- Flask-Bootstrap

Flask-RESTful

在前面，我们介绍了 REST Web 服务，并使用 Flask 提供服务。这里，我们使用第三方库 Flask-RESTful，它使得在 Flask 中提供 REST 服务变得更加简单。

安装

使用 pip 安装：

```
$ pip install flask-restful
```

使用

下面我们主要使用官方文档的例子进行说明。

Hello World

我们先来看一个简单的例子。

```
# -*- coding: utf-8 -*-

from flask import Flask
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

class HelloWorld(Resource):
    def get(self):
        return {'hello': 'world'}

api.add_resource(HelloWorld, '/')

if __name__ == '__main__':
    app.run(debug=True)
```

上面的代码应该不难看懂。我们定义了一个 `HelloWorld` 的类，该类继承自 `Resource` ，在类里面，我们定义了 `get` 方法，该方法跟 HTTP 请求中的 `GET` 方法对应。接着，我们使用 `add_resource()` 方法添加资源，该方法的第 1 个参数就是我们定义的类，第 2 个参数是 URL 路由。

将上面的代码保存为 `app.py` ，在终端运行：

```
$ python app.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 109-566-036
```

使用 `curl` 测试，如下：

```
$ curl http://127.0.0.1:5000/
{
    "hello": "world"
}
```

另外， `add_resource()` 方法也支持添加多个路由，比如：

```
api.add_resource(HelloWorld, '/', '/hello')
```

这样访问 `http://127.0.0.1:5000/` 和 `http://127.0.0.1:5000/hello` 效果是一样的。

带参数的请求

Flask-RESTful 也支持路由带参数的请求，跟 Flask 的实现是类似的，看下面这个例子。

```
# -*- coding: utf-8 -*-

from flask import Flask, request
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

todos = {}

class TodoSimple(Resource):
    def get(self, todo_id):
        return {todo_id: todos[todo_id]}

    def put(self, todo_id):
        todos[todo_id] = request.form['data']
        return {todo_id: todos[todo_id]}

# 路由带参数
api.add_resource(TodoSimple, '/<string:todo_id>')

if __name__ == '__main__':
    app.run(debug=True)
```

使用 curl 测试，如下：

```
$ curl -X PUT http://localhost:5000/todo1 -d "data=shopping"
{
    "todo1": "shopping"
}

$ curl -X GET http://localhost:5000/todo1
{
    "todo1": "shopping"
}
```

参数解析

Flask-RESTful 提供了 `reqparse` 库来简化我们访问并验证表单的工作，将上面的代码使用 `reqparse` 改写，如下：

```
# -*- coding: utf-8 -*-

from flask import Flask, request
from flask_restful import Resource, Api, reqparse

app = Flask(__name__)
api = Api(app)

parser = reqparse.RequestParser()
parser.add_argument('task', type=str)  # 相当于添加 form 表单字段，并给出类型

todos = {}

class TodoSimple(Resource):
    def get(self, todo_id):
        return {todo_id: todos[todo_id]}

    def put(self, todo_id):
        args = parser.parse_args()   # 获取表单的内容, args 是一个字典

        if args['task']:
            todos[todo_id] = args['task']
        else:
            return 'error!'

        return {todo_id: todos[todo_id]}

api.add_resource(TodoSimple, '/<string:todo_id>')

if __name__ == '__main__':
    app.run(debug=True)
```

上面的代码中，我们使用 `add_argument()` 方法添加 form 表单字段，并指定其类型。获取表单内容使用 `parse_args()` 方法，该方法返回一个字典，字典的 key 就是表单的字段。

使用 curl 测试，如下：

```
$ curl -X PUT http://localhost:5000/todo1 -d "task=shopping"
{
    "todo1": "shopping"
}
```

一个完整的例子

下面这个完整的例子来自于官方文档。

```
from flask import Flask
from flask_restful import reqparse, abort, Api, Resource

app = Flask(__name__)
api = Api(app)

TODOS = {
    'todo1': {'task': 'build an API'},
    'todo2': {'task': '?????'},
    'todo3': {'task': 'profit!'},
}

def abort_if_todo_doesnt_exist(todo_id):
    if todo_id not in TODOS:
        abort(404, message="Todo {} doesn't exist".format(todo_i
d))

parser = reqparse.RequestParser()
parser.add_argument('task')

# Todo
# shows a single todo item and lets you delete a todo item
class Todo(Resource):
    def get(self, todo_id):
        abort_if_todo_doesnt_exist(todo_id)
        return TODOS[todo_id]

    def delete(self, todo_id):
        abort_if_todo_doesnt_exist(todo_id)
        del TODOS[todo_id]
        return '', 204
```

```
    def put(self, todo_id):
        args = parser.parse_args()
        task = {'task': args['task']}
        TODOS[todo_id] = task
        return task, 201

# TodoList
# shows a list of all todos, and lets you POST to add new tasks
class TodoList(Resource):
    def get(self):
        return TODOS

    def post(self):
        args = parser.parse_args()
        todo_id = int(max(TODOS.keys()).lstrip('todo')) + 1
        todo_id = 'todo%i' % todo_id
        TODOS[todo_id] = {'task': args['task']}
        return TODOS[todo_id], 201

## Actually setup the Api resource routing here
api.add_resource(TodoList, '/todos')
api.add_resource(Todo, '/todos/<todo_id>')

if __name__ == '__main__':
    app.run(debug=True)
```

运行上面的代码，然后使用 curl 进行测试，如下：

```
# 获取所有 todos
$ curl http://localhost:5000/todos
{
    "todo1": {
        "task": "build an API"
    },
    "todo2": {
        "task": "?????"
    },
    "todo3": {
        "task": "profit!"
    }
}
```

```
# 获取单个 todo
$ curl http://localhost:5000/todos/todo2
{
    "task": "?????"
}

# 删除某个 todo
$ curl -v -X DELETE http://localhost:5000/todos/todo2
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 5000 (#0)
> DELETE /todos/todo2 HTTP/1.1
> Host: localhost:5000
> User-Agent: curl/7.43.0
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 204 NO CONTENT
< Content-Type: application/json
< Content-Length: 0
< Server: Werkzeug/0.11.11 Python/2.7.11
< Date: Tue, 18 Oct 2016 04:02:17 GMT
<
* Closing connection 0

# 添加 todo
$ curl -v -X POST http://localhost:5000/todos -d "task=eating"
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 5000 (#0)
> POST /todos HTTP/1.1
> Host: localhost:5000
> User-Agent: curl/7.43.0
> Accept: */*
> Content-Length: 11
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 11 out of 11 bytes
* HTTP 1.0, assume close after body
< HTTP/1.0 201 CREATED
< Content-Type: application/json
< Content-Length: 25
< Server: Werkzeug/0.11.11 Python/2.7.11
```

```
< Date: Tue, 18 Oct 2016 04:04:16 GMT
<
{
    "task": "eating"
}
* Closing connection 0

# 更新 todo
$ curl -v -X PUT http://localhost:5000/todos/todo3 -d "task=runn
ing"
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 5000 (#0)
> PUT /todos/todo3 HTTP/1.1
> Host: localhost:5000
> User-Agent: curl/7.43.0
> Accept: */*
> Content-Length: 12
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 12 out of 12 bytes
* HTTP 1.0, assume close after body
< HTTP/1.0 201 CREATED
< Content-Type: application/json
< Content-Length: 26
< Server: Werkzeug/0.11.11 Python/2.7.11
< Date: Tue, 18 Oct 2016 04:05:52 GMT
<
{
    "task": "running"
}
* Closing connection 0
```

本文完整的代码在这里。

更多阅读

- Flask-RESTful 0.2.1 documentation
- Designing a RESTful API using Flask-RESTful - miguelgrinberg.com
- Flask扩展系列(一)–Restful – 思诚之道

Flask 实战

完整的项目代码，请参考这里。

本章主要介绍如何开发一个 `Web TODO` 应用，用于管理个人的任务清单，该项目改编自 flask-simple-todo，由于原项目已经很久没更新了，我对其进行了修改和完善，修改后的代码在这里。

目前，该应用主要的功能有：

- 添加待办事项
- 修改待办事项
- 删除事项
- 完成事项

界面如下：

127.0.0.1:7345

TODO Lists

Todo

1. eat apple delete | update | complete
2. play football delete | update | complete

Completed

1. ~~shopping~~ delete | redo
2. ~~have a rest~~ delete | redo

Add

add todo item Submit

Powered by: Flask.

开始实战

我们会从下面三个方面对该项目进行介绍：

1. 生成项目结构
2. 设计数据模型
3. 编写业务逻辑

项目结构规范

我们在前面所举的例子基本都是写在一个单一的脚本文件中，比如 `app.py` ，这在做一些简单的测试中是可行的，但是在较大的项目中则不应该这么做。好的项目结构可以让人更易于查找代码，也易于维护。当然了，每个团队都有自己的项目规范，在这里，我分享自己在平时的开发中经常用到的项目结构，仅供参考。

我们以该 TODO 项目为例，介绍项目结构。

为了方便，这里使用 shell 脚本生成项目基础骨架：

```
# !/bin/bash

dirname=$1

if [ ! -d "$dirname" ]
then
    mkdir ./$dirname && cd $dirname
    mkdir ./application
    mkdir -p ./application/{controllers,models,static,static/css
,static/js,templates}
    touch {manage.py,requirements.txt}
    touch ./application/{__init__.py,app.py,configs.py,extension
s.py}
    touch ./application/{controllers/__init__.py,models/__init__
.py}
    touch ./application/{static/css/style.css,templates/404.html
,templates/base.html}
    echo "File created"
else
    echo "File exists"
fi
```

将上面的脚本保存为文件 `generate_flask_boilerplate.sh` ，使用如下命令生成项目骨架：

```
$ sh generate_flask_boilerplate.sh flask-todo-app
```

生成的项目骨架如下所示：

```
flask-todo-app
├── application
│   ├── __init__.py
│   ├── app.py
│   ├── configs.py
│   ├── controllers
│   │   ├── __init__.py
│   ├── extensions.py
│   ├── models
│   │   ├── __init__.py
│   ├── static
│   │   ├── css
│   │   │   └── style.css
│   │   └── js
│   └── templates
│       ├── 404.html
│       ├── base.html
├── manage.py
├── requirements.txt
```

该项目骨架包含三个顶级文件(夹)：application 目录、manage.py 文件和 requirements.txt 文件，在一般情况下，我们可能还需要一个 tests 目录，存放单元测试的代码，在这里，我们没有把它包含进来。下面，我解释一下该项目骨架：

- application 目录存放 Flask 程序，包含业务逻辑代码、数据模型和静态文件等
    - configs.py 存放项目配置
    - models 目录存放数据模型文件
    - templates 目录存放模板文件
    - static 目录用于存放静态文件，如 js、css 等文件
- manage.py 用于启动我们的 Web 程序以及其他的程序任务
- requirements.txt 文件列出了项目的安装依赖包，便于在其他机器部署

数据模型

该项目是一个 TODO 应用，可以对任务清单进行增加、修改、删除等，相应地，我们需要设计一个数据模型来存储相应的数据和状态。不难想到，表的字段主要有以下几个：

- id: 标识每条记录的字段，是表的主键，Integer 类型；
- title: 即任务清单的标题，String 类型；
- posted_on: 任务创建时间，DATE 类型；
- status: 任务的状态，Boolean 类型；

因此，我们的数据模型定义如下 (完整代码参考这里)：

```
# -*- coding: utf-8 -*-

from application.extensions import db
from datetime import datetime

__all__ = ['Todo']

class Todo(db.Model):
    """数据模型"""
    __tablename__ = 'todo'

    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(255), nullable=False)
    posted_on = db.Column(db.Date, default=datetime.utcnow)
    status = db.Column(db.Boolean(), default=False)

    def __init__(self, *args, **kwargs):
        super(Todo, self).__init__(*args, **kwargs)

    def __repr__(self):
        return "<Todo '%s'>" % self.title
```

创建数据库

这里，我们使用 Flask-Migrate 插件管理数据库的迁移和升级，详细用法可以查看 Flask-Migrate。

在项目的根目录下按顺序执行如下命令，生成 migratios 文件夹、数据迁移脚本和更新数据库。

```
$ python manage.py db init   # 初始化，生成 migratios 文件夹
  Creating directory /Users/admin/flask-todo/migrations ... done
  Creating directory /Users/admin/flask-todo/migrations/versions
 ... done
  Generating /Users/admin/flask-todo/migrations/alembic.ini ...
done
  Generating /Users/admin/flask-todo/migrations/env.py ... done
  Generating /Users/admin/flask-todo/migrations/env.pyc ... done
  Generating /Users/admin/flask-todo/migrations/README ... done
  Generating /Users/admin/flask-todo/migrations/script.py.mako .
.. done
  Please edit configuration/connection/logging settings in '/Use
rs/admin/flask-
  todo/migrations/alembic.ini' before proceeding.

$ python manage.py db migrate  # 自动创建迁移脚本
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional
DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'todo'
  Generating /Users/admin/flask-todo/migrations/versions/70215a3
905e0_.py ... done

$ python manage.py db upgrade   # 更新数据库
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional
DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> 70215a3905
e0, empty message
```

编写业务逻辑

我们的业务逻辑代码主要在 controllers 目录中，新建一个 `todo.py` 文件，核心代码如下 (完整代码参考这里)，代码说明可以参考注释：

```
# -*- coding: utf-8 -*-

import flask
from flask import request, redirect, flash, render_template, url
_for
from application.extensions import db
from application.models import Todo

todo_bp = flask.Blueprint(
    'todo',
    __name__,
    template_folder='../templates'
)

# 主页
@todo_bp.route('/', methods=['GET', 'POST'])
def index():
    todo = Todo.query.order_by('-id')
    _form = request.form

    if request.method == 'POST':
        # 添加任务
        title = _form["title"]
        td = Todo(title=title)
        try:
            td.store_to_db()  # 将数据保存到数据库
            flash("add task successfully!")
            return redirect(url_for('todo.index'))
        except Exception as e:
            print e
            flash("fail to add task!")

    return render_template('index.html', todo=todo)

# 删除任务
```

```
@todo_bp.route('/<int:todo_id>/del')
def del_todo(todo_id):
    todo = Todo.query.filter_by(id=todo_id).first()
    if todo:
        todo.delete_todo()
    flash("delete task successfully")
    return redirect(url_for('todo.index'))

# 编辑 (更新) 任务
@todo_bp.route('/<int:todo_id>/edit', methods=['GET', 'POST'])
def edit(todo_id):
    todo = Todo.query.filter_by(id=todo_id).first()

    if request.method == 'POST':
        Todo.query.filter_by(
            id=todo_id
        ).update({
            Todo.title: request.form['title']
        })
        db.session.commit()
        flash("update successfully!")
        return redirect(url_for('todo.index'))

    return render_template('edit.html', todo=todo)

# 标记任务完成
@todo_bp.route('/<int:todo_id>/done')
def done(todo_id):
    todo = Todo.query.filter_by(id=todo_id).first()
    if todo:
        Todo.query.filter_by(id=todo_id).update({Todo.status: Tr
ue})
        db.session.commit()
        flash("task is completed!")

    return redirect(url_for('todo.index'))

# 重置任务
@todo_bp.route('/<int:todo_id>/redo')
def redo(todo_id):
    todo = Todo.query.filter_by(id=todo_id).first()
    if todo:
```

```
        Todo.query.filter_by(id=todo_id).update({Todo.status: False})
        flash("redo successfully!")
        db.session.commit()

    return redirect(url_for('todo.index'))

# 404 处理
@todo_bp.errorhandler(404)
def page_not_found():
    return render_template('404.html'), 404
```

上面只是核心的业务逻辑，关于模板渲染，数据模型的定义等可以参考完整代码。

由于该项目的业务逻辑比较简单，因此也就不做过多介绍，相信读者可以轻松看懂代码，如果有问题，也可以在 github 上面提 issue，欢迎指正。

结束语

本书介绍了如何基于 Flask 进行 Web 开发，涉及的内容也比较基础，相信读者看完本书后可以做一个简单的 Web 应用。本书的实战例子是一个 Web TODO 应用，由后台进行模板渲染返回给客户端，读者如果有兴趣的话，可以对其进行改编，比如将前端和后端分离，后端负责数据处理，前端负责渲染，如果你对 React 感兴趣的话，可以考虑一下。

本书还有很多其他议题没有涉及到，比如测试，性能等等。另外，如果本书存在什么问题，请与我联系，欢迎指正。

参考链接

本书主要参考资源：

- Flask入门系列 - 思诚之道
- Flask扩展系列 - 思诚之道
- Flask进阶系列 – 思诚之道
- 浅入浅出 Flask 框架
- flask 之旅
- flask docs
- flask-mega-tutorial
- wwq0327 / flask-simple-todo
- FlaskBook.com

flask 资源推荐

这里列出个人在网上找到的一些不错的资源，欢迎读者补充。

- humiaozuzu/awesome-flask: A curated list of awesome Flask resources and plugins
- miguelgrinberg/flasky: Companion code to my O'Reilly book "Flask Web Development"
- sh4nks/flaskbb: A forum software written in flask
- Flask 大型教程项目 — flask mega-tutorial 1.0 文档